PHP

Clube Delphi +PHP

Ano 8 · Edição 102 · R$11,90

DevMedia group

Orientação a Objetos no PHP

Desenvolva Web Sites inteligentes e totalmente OO com PHP

Relatório Web

Crie relatórios na Web em PDF com o PHP

MVC

Do conceito a prática

Ask the Expert

InputBox com ComboBox

QuickUpdate

Instale componentes facilmente com o DelphiPI

Boa Idéia

Crie um sistema de chat tipo Messenger e melhore o suporte a clientes

Automação de Testes

Aplique testes unitários, funcionais e não funcionais em sua aplicação

Compre esta edição e ganhe:

Confira no portal ClubeDelphi PLUS, 4 vídeo-aulas:

- Trabalhando com processos do Windows
- Gerenciando configurações de sistema
- Segredos do DBGrid
- Mais conteúdo para o Treeview

Easy Delphi

Saiba criar e usar dlls em suas aplicações

Automatize tarefas criando serviços para Windows

ISSN 1517990-7

Já pensou em hospedar todos os seus sites em um único plano de hospedagem?

PLANO PROFISSIONAL 1

- 3 domínios independentes
- 50 GB de transferência
- 30 GB de e-mails com SSL
- 1 GB de espaço
- Painel de controle
- ASP, ASP.NET 3.5 E PHP 5
- Access ilimitado
- 3 bancos MYSQL 5
- 1 banco SQL Server 2005 Express

30 DIAS GRÁTIS

Ao assinar, digite o código de desconto exclusivo para leitores desta revista e ganhe!

RVSTMEDIA

Tudo isso por apenas R$ **25,90** por mês

Na Hospedix você ainda ganha registro de domínio (.com .net .org ou .info) grátis e isento de taxas anuais enquanto for cliente.

Acesse agora mesmo e descubra por que Hospedix é a hospedagem de sites preferida entre os profissionais.

hospedix.com.br

HOSPEDIX
HOSPEDAGEM PROFISSIONAL

Sumário

Olá, eu sou o **DevMan**! Desta página em diante, eu estarei lhe ajudando a compreender com ainda mais facilidade o conteúdo desta edição. Será um prazer contar com sua companhia! Confira abaixo o que teremos nesta revista:

Boas Práticas

Boas Práticas **Projeto/Análise**

14 – TestComplete
Aplique testes unitários, funcionais e não funcionais em sua aplicação
[*Cristiano Caetano*]

Win32 **Boa Idéia**

20 – Messenger Personalizado
Crie um aplicativo tipo Messenger para suporte técnico
[*Paulo Quicoli*]

Boas Práticas **Projeto/Análise**

32 – Aplicações Win32 com MVC
Aplique o poderoso conceito MVC a aplicações Win32
[*Rodrigo Mourão*]

PHP

PHP

42 – Emitindo relatórios em PDF
Veja como emitir relatórios em PDF para PHP
[*Fabrício Desbessel*]

PHP

48 – Conceitos OO e novidades do PHP
Aprenda os principais conceitos OO e veja as novidades do PHP 5.3
[*Alexandre Altair*]

Easy Delphi

Easy Delphi

54 – Services Applications
Automatize tarefas desenvolvendo serviços do Windows
[*Maikel Scheid*]

PHP

62 – Desenvolvendo DLL's
Aprenda a desenvolver e usar dll's em seus sistemas
[*Adriano Santos*]

Você percebeu os nomes ao lado de cada matéria? Eles indicam o que você vai encontrar no artigo – dessa forma, você também pode ter uma idéia geral do que vai encontrar nesta edição como um todo! Os editores trabalham sempre no sentido de fechar a revista seguindo esta definição, para oferecer a você o melhor conteúdo didático!

Confira abaixo a lista com a definição dos tipos de artigo encontrados nesta edição:

[Mão na Massa] Artigos que focam na resolução de problemas - e não na tecnologia. A tecnologia empregada é apenas conseqüência. O artigo parte do pressuposto que o leitor já conhece a tecnologia, mas tem dificuldade de implementá-la em uma aplicação cotidiana. Por exemplo, um artigo que ao invés de "debulhar" todos os métodos e propriedades de um "Recordset", mostre como usar o Recordset de forma inteligente. para criar um carrinho de compras.

[PHP] Artigo com foco no PHP puro.

[Expert] Artigo com foco no leitor avançado.

[Boas Práticas] A Um dos objetivos da revista é levar para o leitor não somente as melhores técnicas, mas também as melhores práticas – Esse tipo de artigo foca em técnicas que poderão aumentar a qualidade do desenvolvimento de software.

[Web] Artigos sobre ou que envolvam técnicas de desenvolvimento para WEB.

[Novidades] Artigos sobre tecnologias de ponta, que ainda não fazem parte do dia a dia do desenvolvedor mas que prometem ser a próxima febre.

[Boa Idéia] A Esse tipo de artigo é quase um artigo "Mão na Massa" - a diferença é que o exemplo do artigo é tão criativo que é considerado uma 'boa idéia', dentro do contexto de desenvolvimento de software.

[Easy Delphi] Com foco no desenvolvedor iniciante.

Clube Delphi +PHP

Ano 8 - 102ª Edição - 2008 - ISSN 1517990-7 **Impresso no Brasil**

Corpo Editorial

Editor Geral

Adriano Santos

adrianosantos@devmedia.com.br

Editor Técnico

Paulo Quicoli

pauloquicoli@devmedia.com.br

Comissão Editorial

Fabrício Desbessel, Paulo Quicoli, Adriano Santos e Guinther Pauli.

Editor de Arte

Vinicius O. Andrade

viniciusoandrade@gmail.com

Revisão

Gregory Monteiro

gregory@clubedelphi.net

Distribuição

Fernando Chinaglia Dist. S/A

Rua Teodoro da Silva, 907

Grajaú - RJ - 206563-900

Atendimento ao Leitor

A DevMedia conta com um departamento exclusivo para o atendimento ao leitor. Se você tiver algum problema no recebimento do seu exemplar ou precisar de algum esclarecimento sobre assinaturas, exemplares anteriores, endereço de bancas de jornal, entre outros, entre em contato com:

Carmelita Mulin
www.devmedia.com.br/central/default.asp
(21) 3382-5025

Kaline Dolabella
Gerente de Marketing e Atendimento
kalined@terra.com.br
(21) 3382-5025

Publicidade

Para informações sobre veiculação de anúncio na revista ou no site entre em contato com:

Kaline Dolabella
publicidade@devmedia.com.br

Fale com o Editor

É muito importante para a equipe saber o que você está achando da revista: que tipo de artigo você gostaria de ler, que artigo você mais gostou e qual artigo você menos gostou. Fique a vontade para entrar em contato com os editores e dar a sua sugestão!

Se você estiver interessado em publicar um artigo na revista ou no site ClubeDelphi, entre em contato com os editores, informando o título e mini-resumo do tema que você gostaria de publicar:

Guinther Pauli - Editor da Revista
guinther@devmedia.com.br

EDITORIAL

É com enorme satisfação que escrevo esse editorial. Cresci na carreira de desenvolvedor Delphi lendo a revista ClubeDelphi que com certeza foi uma principais responsáveis pelo meu sucesso profissional. Hoje, assumir a edição geral da revista que me formou profissional, sem dúvida me proporcionará muito prazer e alegria. Com muito respeito e admiração, agradeço aos amigos Guinther Pauli e Gladstone Matos pelas oportunidades e confiança no meu trabalho.

A revista desse mês vem recheada de muito conteúdo de qualidade, como nosso artigo de capa. O Rodrigo Mourão fala sobre a aplicação de MVC – Model View Controller - em sistemas Win32. Certamente, é um poderoso conceito para quem deseja desenvolver programas robustos e de peso, totalmente OO.

Outro tema importante é abordado pelo Paulo. Imagine criar seu próprio Messenger e melhorar ainda mais a comunicação de seus colaboradores com seus clientes? Pois é essa a proposta do Paulo Quicoli que mostra como desenvolver uma aplicação tipo Messenger para prover suporte técnico aos clientes. Na seqüência o Cristiano apresenta a ferramenta TestComplete, um poderoso framework para testes de aplicações.

Na sessão PHP temos o Fabrício Desbessel mostrando como desenvolver relatórios completos usando o componente FPDF e também o Altair com seu artigo sobre conceitos de Orientação a Objetos e novidades do novo PHP 5.3.

Na sessão EasyDelphi temos dois grandes artigos. Em meu artigo sobre o desenvolvimento e uso de DLL, você aprenderá a criar funcionalidades em dll's e usá-las em suas aplicações. Ainda será possível ver como se carrega uma dll de modo estático ou dinâmico. O Maikel Scheid fala sobre o desenvolvimento de Services Applications, ou Aplicações Serviço. Essas aplicações ficam residentes na memória do computador e podem executar tarefas mesmo quando o Windows encontra-se no prompt de login. É ideal para sistemas que necessitam fazer monitoramentos, acesso constante ao banco de dados ou até mesmo fazer a automatização de backups e atualizações do sistema.

Sucesso com o Delphi! E como diria Guinther Pauli: muito Delphi na cabeça!

Adriano Santos
adrianosantos@devmedia.com.br
Editor Chefe da revista ClubeDelphi
Conheça nosso BLOG em clubedelphiphp.blogspot.com

ClubeDelphi PLUS

DevMedia group

A revista ClubeDelphi é parte integrante da assinatura ClubeDelphi PLUS.
Para mais informações sobre o pacote PLUS, acesse:
http://www.devmedia.com.br/clubedelphi/portal.asp

Hospede com quem fala sua língua!

Local host

Hospedagem Profissional
Revenda
Parcerias

Ativação imediata!

Faça sua assinatura, tenha 30 dias grátis

e seu primeiro boleto para 60 dias após assinatura em qualquer plano

UNIX I

a partir R$ 28,00

- 1 GB Espaço em disco
- 60 GB Transferência
- 30 Contas de e-mail
- 10 Domínios
- Painel de Controle em Português
- Atendimento exclusivo

Contratar

UNIX II

a partir R$ 58,00

- 3 GB Espaço em disco
- 120 GB Transferência
- 60 Contas de e-mail
- 15 Domínios
- Painel de Controle em Português
- Atendimento exclusivo

Contratar

UNIX III

a partir R$ 98,00

- 6 GB Espaço em disco
- 300 GB Transferência
- 90 Contas de e-mail
- 20 Domínios
- Painel de Controle em Português
- Atendimento exclusivo

Contratar

R$ 5,90 — Mini html

Tráfego 1 GB
Espaço 100 MB
3 C.Ppostal, Espaço 3G
Domínio 3

R$ 15,90 — html

Tráfego 25 GB
Espaço 500 MB
20 C.Ppostal, Espaço 20G
Domínio Ilimitado

R$ 15,90 — ASP

Tráfego 25 GB
Espaço 500 MB
20 C.Ppostal, Espaço 20G
Domínio Ilimitado
MySql 50m

R$ 20,90 — PHP

Tráfego 25 GB
Espaço 500 MB
20 C.Ppostal, Espaço 20G
Domínio Ilimitado
MySql 50m

Formato exclusivo de trabalho para empresas de desenvolvimento de sites, websites, portais, serviços e negócios que necessitam de grandes tráfegos, monitoramento e backup.

www.lhost.com.br

assinar@lhost.com.br

Assinatura

ClubeDelphi PLUS

Mais conteúdo Delphi por menos

Informativo ClubeDelphi

Portal ClubeDelphi

www.clubedelphi.net/portal

+600 vídeo aulas e 7 cursos online

Brinde na web desta edição

4 Vídeos

Trabalhando com processo no Windows
Nessa vídeo-aula o autor Guinther Pauli mostra como monitorar e mostrar os processos que estão rodando na máquina em um Grid.
http://www.devmedia.com.br/articles/viewcomp.asp?comp=4406

Gerenciando configurações de sistema – Parte 1
http://www.devmedia.com.br/articles/viewcomp.asp?comp=4660

Segredos do DBGrid – Parte 1
http://www.devmedia.com.br/articles/viewcomp.asp?comp=6328

Mais conteúdo para o Treview
http://www.devmedia.com.br/articles/viewcomp.asp?comp=6348

Para acessar os vídeos utilize os seguintes dados: Login: **DVM.PLS** Senha: **k9b7u**

Gostou das vídeo aulas? O portal *www.devmedia.com.br* possui mais de **2 mil vídeo aulas** e **dezenas de cursos online** sobre desenvolvimento de software! Agora você pode comprar as vídeo aulas que preferir e fazer sua própria combinação de vídeos! Saiba mais em *www.devmedia.com.br/creditos*

Edições Anteriores da ClubeDelphi

Você pode comprar todas as edições anteriores da ClubeDelphi através do site DevMedia!

Para isso basta acessar *https://seguro.devmedia.com.br/edicoes_anteriores.asp*

Dê seu feedback sobre esta edição!

A ClubeDelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre esta edição, artigo por artigo, através do link: *www.devmedia.com.br/clubedelphi/feedback*

Para votar, você vai precisar do código e senha de banca desta edição, que é:

Login: **DVM.PLS** Senha: **k9b7u**

Você não está mais sozinho.

A partir de agora você pode contar com a ajuda da

Chegou a Consultoria On-line DevMedia

Consultoria Técnica + Professor Virtual + Certificação

Mais Informações:
www.devmedia.com.br/consultoria_online 21 3382-5025

DevMedia em seus projetos e estudos.

A DevMedia possui um numeroso time de autores, editores e professores que juntos produzem o material que você está acostumado a encontrar em nosso site e revistas. E são exatamente esses mesmos profissionais que estarão a sua disposição para tirar suas dúvidas e ajudá-lo em seus projetos e estudos. Através de uma plataforma 100% web a Consultoria DevMedia garante sigilo absoluto, eficiência e rapidez em todas as respostas. Finalmente você terá ao seu alcance uma consultoria de qualidade por um preço muito acessível. Consulte nossos planos.

Mais um serviço DevMedia group

Nesta seção você encontra artigos intermediários sobre Delphi Win32 e Delphi for .NET

Perguntas e Respostas

Convertendo nomes de arquivos em nomes longos ou curtos

Perguntas respondidas por Paulo Quicoli
pauloquicoli@devmedia.com.br

A poucos dias do fechamento dessa edição, recebi um e-mail de um colega que me pediu para ajudá-lo a criar um *InputQuery* que tivesse um *ComboBox* no lugar do *Edit*. Muito bem, já havia feito meu próprio *InputQuery* com s enha, mas não com *ComboBox*. Fiz uma rápida pesquisa na internet e acabei encontrando uma solução bem legal. Primeiro criamos uma *function* que será a nossa *InputBox*. Veja a **Listagem 1.** Em seguida fazemos incluímos um *Label* e um *Button* na tela para que possamos testar a solução. No código do evento *OnClick* no *Button* insira o código da **Listagem 2**. Execute e teste. Veja que temos itens no *ComboBox* e ainda por cima ao clicar em um deles o *Label* que foi colocado é atualizado com o item que foi selecionado. Veja nas **Figura 1** e **2** o esquema em execução. ●

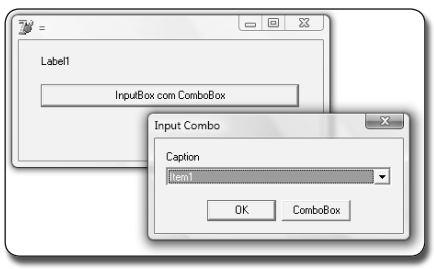


Figura 1. InputCombo em ação

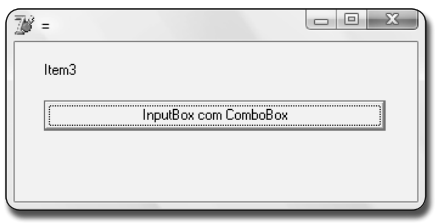


Figura 2. Resultado da seleção do Combo

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

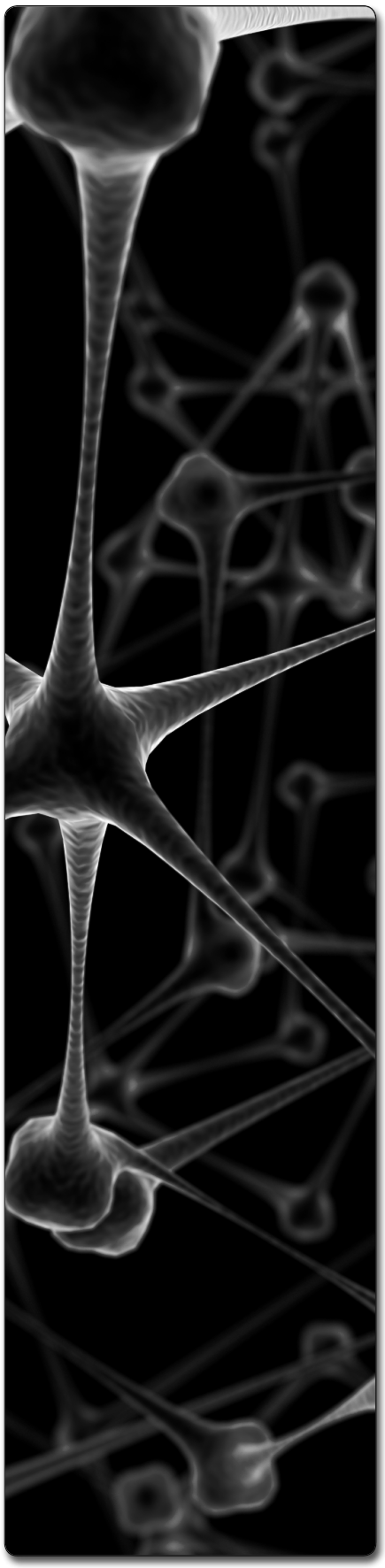
www.devmedia.com.br/clubedelphi/feedback

Listagem 1. Código do InputBox personalizado com ComboBox

```
function TForm1.InputCombo(const ACaption, APrompt: string; const AList: TStrings):
string;
  function GetCharSize(Canvas: TCanvas): TPoint;
  var
    I: Integer;
    Buffer: array[0..51] of Char;
  begin
    for I := 0 to 25 do Buffer[I] := Chr(I + Ord('A'));
    for I := 0 to 25 do Buffer[I + 26] := Chr(I + Ord('a'));
    GetTextExtentPoint(Canvas.Handle, Buffer, 52, TSize(Result));
    Result.X := Result.X div 52;
  end;
var
  Form: TForm;
  Prompt: TLabel;
  Combo: TComboBox;
  DialogUnits: TPoint;
  ButtonTop, ButtonWidth, ButtonHeight: Integer;
begin
  Result := '';
  Form   := TForm.Create(Application);
  with Form do
    try
      Canvas.Font := Font;
      DialogUnits := GetCharSize(Canvas);
      BorderStyle := bsDialog;
      Caption     := ACaption;
      ClientWidth := MulDiv(180, DialogUnits.X, 4);
      Position    := poScreenCenter;
      Prompt      := TLabel.Create(Form);
      with Prompt do
      begin
        Parent   := Form;
        Caption  := APrompt;
        Left     := MulDiv(8, DialogUnits.X, 4);
        Top      := MulDiv(8, DialogUnits.Y, 8);
        Constraints.MaxWidth := MulDiv(164, DialogUnits.X, 4);
        WordWrap := True;
      end;
      Combo := TComboBox.Create(Form);
      with Combo do
      begin
        Parent := Form;
        Style  := csDropDownList;
        Items.Assign(AList);
        ItemIndex := 0;
        Left      := Prompt.Left;
        Top       := Prompt.Top + Prompt.Height + 5;
        Width     := MulDiv(164, DialogUnits.X, 4);
      end;
      ButtonTop    := Combo.Top + Combo.Height + 15;
      ButtonWidth  := MulDiv(50, DialogUnits.X, 4);
      ButtonHeight := MulDiv(14, DialogUnits.Y, 8);
      with TButton.Create(Form) do
      begin
        Parent      := Form;
        Caption     := 'OK';
        ModalResult := mrOk;
        default     := True;
        SetBounds(MulDiv(38, DialogUnits.X, 4), ButtonTop, ButtonWidth,
          ButtonHeight);
      end;
      with TButton.Create(Form) do
      begin
        Parent      := Form;
        Caption     := 'ComboBox';
        ModalResult := mrCancel;
        Cancel      := True;
        SetBounds(MulDiv(92, DialogUnits.X, 4),
          Combo.Top + Combo.Height + 15,
          ButtonWidth, ButtonHeight);
        Form.ClientHeight := Top + Height + 13;
      end;
      if ShowModal = mrOk then
      begin
        Result := Combo.Text;
      end;
    finally
      Form.Free;
    end;
end;
```

Listagem 2. Chamada ao InputBox personalizado

```
procedure TForm1.Button1Click(Sender: TObject);
var
  List: TStringList;
begin
  List := TStringList.Create;
  try
    List.Add('Item1');
    List.Add('Item2');
    List.Add('Item3');
    Label1.Caption := InputCombo('Input Combo', 'Caption', List);
  finally
    List.Free;
  end;
end;
```



Nesta seção você encontra artigos intermediários sobre Delphi Win32 e Delphi for .NET

Quick Update

Instalando componentes mais facilmente no Delphi

Adriano Santos
(falecom@adrianosantos.pro.br)
é desenvolvedor Delphi desde 1998, professor e programador PHP, Bacharel em Comunicação Social pela Universidade Cruzeiro do Sul, SP, editor chefe da revista ClubeDelphi, mantém o blog Delphi to Delphi (www.delphitodelphi.blogspot.com) com dicas, informações e tudo sobre desenvolvimento Delphi e é mebro fundador do DUG-SP – Delphi Users Group São Paulo.

Olá, nessa edição o mérito todo vai para o colega e leitor Sérgio Guedes (*borlandlinuxer@gmail.com*). Ele nos mandou um artigo interessante sobre uma ferramenta chamada *DelphiPI*. O principal objetivo dessa ferramenta é facilitar a instalação de componentes no Delphi. Para quem está pouco acostumado ou simplesmente utiliza um bom volume de componentes em sua aplicação, o *DelphiPI* vem de encontro a essas necessidades. Seu código-fonte é open source, ou seja, código aberto. É possível efetuar o download dele no site *delphipi.googlecode.com/*. Vamos ver como a ferramenta funciona.

Baixe a versão 0.40 do site indicado em um local a sua escolha. Após efetuado o download execute a aplicação que solicitar a preferência de linguagem, confirme e aguarde o início da instalação. Prossiga confirmando cada tela de instalação, ao final marque a opção *Launch DelphiPI* e finalize.

Para demonstrar o *DelphiPI* vamos instalar um componente qualquer em nossa IDE. No meu caso vou utilizar comum para validação de *CGC/CPF*.

Ao abrirmos o *DelphiPI*, notamos uma tela bastante simples como pode ser visto na **Figura 1**.

O próximo passo é informar o caminho do arquivo do componente que se deseja instalar através do campo *Base Folder* no grupo *Select Base Folder contains both Package and Source Files*. O campo *Base Folder* é o campo onde informamos a base dos arquivos de um pacote, por exemplo: existem componente na internet que possuem uma série de arquivos *.pas* associados a ele, e por isso diversas pastas fazem parte de seu pacote.

Em seguida, precisamos informar o *Package File Pattern* do componente, ou seja, a extensão do arquivo que contém nosso componente, por padrão *.dpk*. Clique em *Next* para continuar.

Na próxima etapa, é necessário informarmos em qual Delphi queremos

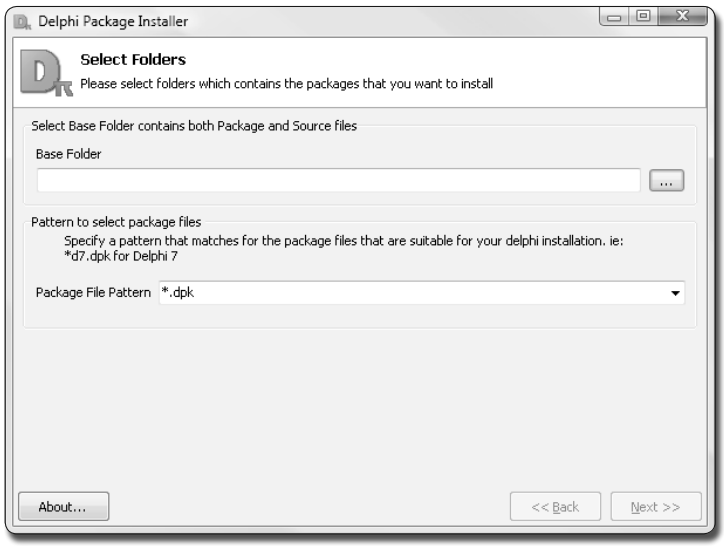


Figura 1. Tela principal do DelphiPI

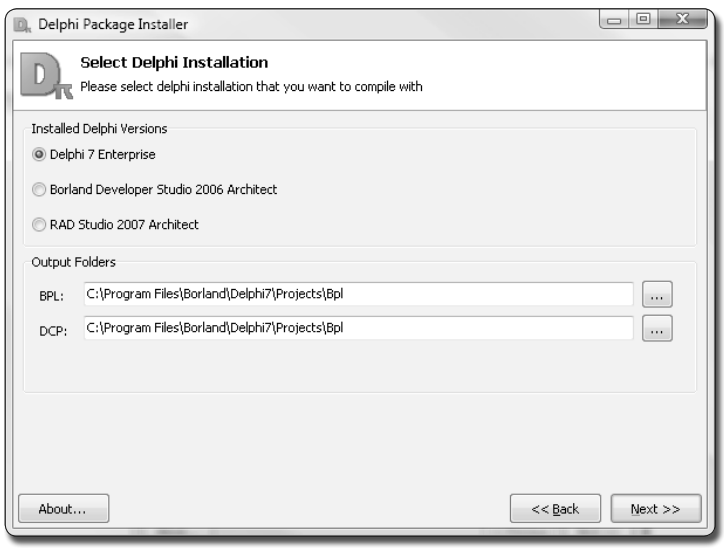


Figura 2. Escolhendo o Delphi

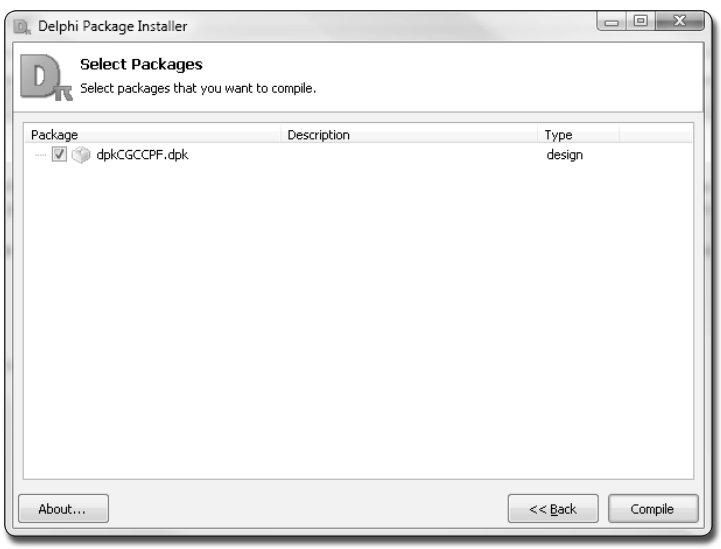


Figura 3. Lista de pacotes encontrados pelo DelphiPI

fazer a instalação e quais pastas serão gravados os arquivos do componente após a compilação. Note que é possível instalarmos em *Delphi 7*, *BDS 2006* e *RAD Studio 2007* (**Figura 2**). Novamente clique em *Next*.

Automaticamente o *DelphiPI* localiza na pasta informada na primeira tela, que há um pacote de instalação no diretório e o marca como sendo um dos pacotes a instalar. Se tivéssemos mais de um *.dpk* na pastas, os mesmos seriam listados (**Figura 3**). Agora basta clicar em *Compile* e aguardar.

Depois de finalizada a instalação, basta abrir o Delphi que foi escolhido para tal instalação e notar que o componente foi instalado corretamente (**Figura 4**).

Com certeza o *DelphiPI* é uma ferramenta fundamental para agilizar o processo de instalação de componentes. Após receber a dica do leitor, fiz questão de testar diversos pacotes de instalação em diferentes versões do Delphi e realmente funciona muito bem o programa. Note também que a versão 0.40, mencionada nesse artigo, é compatível com a versão 2009 do Delphi. ●

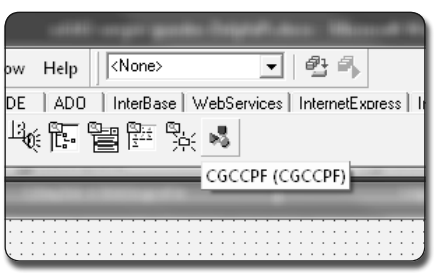


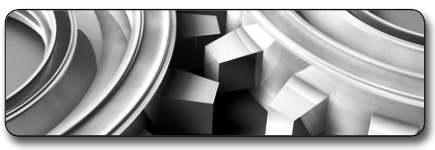
Figura 4. Componente instalado com sucesso

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback



Nesta seção você encontra artigos sobre técnicas que poderão aumentar a qualidade do desenvolvimento de software

Automação de testes

Automatize os testes de seus sistemas usando o software TestComplete

Resumo DevMan

Um assunto pouco explorado hoje em dia ainda está relacionado ao teste de software. Muitas empresas ainda não possuem práticas eficazes para aplicar testes em seus aplicativos ou muitas vezes nem o fazem, deixando somente a cargo do desenvolvedor do sistema. Essa falta de testes acarreta em mau funcionamento do aplicativo quando distribuído.

O que pouca gente sabe é que o mercado está cada vez mais exigente e a qualidade de software cada vez mais é quesito para aquisição de um novo sistema. O grande problema é que uma boa parte das software houses hoje em dia efetuam seus testes manualmente, comprometendo todo o software.

Automatizar o processo de homologação de um novo sistema ou funcionalidade do produto, faz com que o nível de qualidade se eleve e fortaleça o produto como um todo. Nesse artigo veremos como utilizar a ferramenta TestComplete para efetuar testes automatizados em nossas aplicações.

Nesse artigo veremos

• Automação de testes;
• Como utilizar o software TestComplete para automatizar testes;
• A importância de se efetuar testes em aplicações reais.

Qual a finalidade

• Entender a necessidade de se automatizar os testes e apresentar o software TestComplete como uma ferramenta de auxílio nessa automatização;

Quais situações utilizam esses recursos?

• Qualquer software necessita de homologação para se comercializar e distribuir sem maiores transtornos, por isso os testes devem ser realizados em todo tipo de software;

Cristiano Caetano

(cristiano.caetano@testanywhere.com.br)

é certificado CBTS pela ALATS. Diretor da TestAnywhere, consultoria de teste de software. Possui mais de 10 anos de experiência, já trabalhou na área de qualidade e teste de software para grandes empresas como Zero G, DELL e HP Invent. Autor dos livros "CVS: Controle de Versões e Desenvolvimento Colaborativo de Software" e "Automação e Gerenciamento de Testes: Aumentando a Produtividade com as Principais Soluções Open Source e Gratuitas". Criador e mantenedor do portal TestExpert

É muito comum encontrar desenvolvedores e empresas que ainda testam seus softwares apenas de forma manual. E com o aumento de alterações e novas funcionalidades que são incluídas nesses softwares fica difícil manter o ritmo e a qualidade dos testes que são aplicados, comprometendo assim a qualidade final do projeto.

Uma abordagem baseada puramente em testes manuais, normalmente não consegue acompanhar as demandas e o volume de homologações do sistema ao longo do ciclo de vida de desenvolvimento de software. Freqüentemente o sistema é liberado sem que tenha sido completamente testado em virtude de falta de tempo e recursos.

A automação de testes, quando utilizada corretamente, permite a execução ininterrupta de testes a qualquer hora do dia ou da noite. A automatização desse processo é sempre mais rápida do que um processo manual e menos suscetível a erros, afinal, a máquina nunca erra. Neste artigo aprenderemos um pouco sobre os principais paradigmas de automação de testes e as funcionalidades básicas do *TestComplete*, uma ferramenta de automação de testes que tem um ótimo suporte para sistemas desenvolvidos em Delphi.

Paradigmas da automação de testes

Os testes automatizados interagem com o sistema a ser testado. Alguns podem testar um sistema simulando o uso do mesmo por um usuário, como se os controles de um determinado formulário fossem clicados, enquanto que outros aplicam testes em uma parte mais interna do sistema, nas rotinas que formam as regras de negócio. Os tipos de automação são normalmente agrupados de acordo com a forma como os testes automatizados interagem com o sistema. Em geral temos:

• *Baseados na Interface Gráfica*: Nessa abordagem os testes automatizados interagem diretamente com a interface gráfica do sistema simulando um usuário. Normalmente as ações dos usuários são gravadas (*Capture*) por meio da ferramenta. A ferramenta transforma as ações dos usuários em um *script* que pode ser reproduzido (*Playback*) posteriormente.

- *Vantagens*: Não requer modificações no sistema para a automatização. Não é necessário tornar o sistema mais fácil de testar (*testabilidade*) porque eles se baseiam na mesma interface utilizada pelos usuários;
- *Desvantagens*: Existe uma forte dependência da interface gráfica. Se a interface gráfica mudar, os testes falham. Por exemplo, um formulário é criado contendo um *CheckBox* e um teste é criado para verificar seu estado. Se amanhã esse *CheckBox* for substituído por um *RadioButton* o teste irá falhar, caso não seja reconstruído. Baixo desempenho para testes automatizados que exigem centenas de milhares de repetições, testes de funcionalidades que realizam cálculos complexos, integração entre sistemas diferentes e assim por diante.

Item de projeto	Descrição
ActiveX Objects	Este item de projeto permite o reconhecimento de objetos *ActiveX*.
Events	Este item de projeto permite a sobreposição e personalização da ação que é disparada nos eventos ocorridos durante a execução do teste.
HTTP Load Testing	Permite a criação e execução de testes de desempenho de aplicações *WEB*.
Low-Level Procedures Collection	Permite a criação de testes baseados nas coordenadas (X, Y) da tela.
Manual Tests	Criação e execução de casos de testes manuais.
Name Mapping	Este item de projeto permite renomear os objetos para nomes personalizados e mais curtos para facilitar a leitura e criação de *scripts*.
Network Suite	Execução de testes distribuídos em vários computadores.
ODT	Criação de *scripts* de testes dirigidos a objetos (*Object-driven testing*).
Script	Permite a criação e armazenamento dos *scripts* de testes (que podem ser criados nas seguintes linguagens: *VBScript, JScript, DelphiScript, C++ Script ou C# Script).*
Stores	Este item de projeto armazena arquivos que são usados para posterior comparação durante a execução dos testes.
Unit Testing	Este item de projeto permite a criação de testes unitários que podem ser criados nas seguintes tecnologias: *MSTest , JUnit, NUnit, DUnit ou TestComplete unit tests.*
User Forms	Este item de projeto permite a criação de formulários personalizados que podem ser usados para obter informações antes da execução dos testes ou apresentar informações ao longo ou ao final da execução.
Web Services	Este item de projeto permite a criação de testes de *Web Services.*
Win32 Tested Applications	Este item de projeto permite um controle da execução das aplicações que são testadas pelo *TestComplete*.

Tabela 1. Itens de projeto do TestComplete

• *Baseados na Lógica de Negócio*: Nessa abordagem os testes automatizados exercitam as funcionalidades do sistema sem interagir com a interface gráfica. A interface gráfica é apenas uma casca (*camada*) que tem o objetivo de fornecer um meio para a entrada dos dados e apresentação dos resultados. A camada que abriga a funcionalidade e o comportamento do sistema é a camada de lógica de negócio. Essa abordagem de testes é baseada no entendimento que 80% das falhas estão associados a erros na lógica de negócio, ou seja, existem poucos erros críticos na camada de interface com o usuário. Normalmente é necessário realizar modificações no sistema para torná-lo compatível com essa abordagem. Essas modificações resultam em mecanismos para expor ao mundo exterior as funcionalidades internas do sistema.

- *Vantagens*: Foco na camada onde

Nota do DevMan

O *DUnit* é uma ferramenta de código aberto utilizada para efetuar testes unitários em programas desenvolvidos em Pascal ou Delphi. Ele é baseado no *JUnit*, para Java. Porém, a principal premissa para utilização do *DUnit* é que o sistema seja orientado a objetos, diferentemente do *TestComplete* que pode efetuar testes em qualquer tipo de sistema.

Há outros projetos que prometem o mesmo recurso, como o *nUnit*. Uma boa dica é acessar o site *dunit.sourceforge.net/* e entender melhor como funciona a ferramenta como um todo.

ClubeDelphi PLUS www.devmedia.com.br/clubedelphi/portal.asp

Acesse agora mesmo o portal do assinante ClubeDelphi e assista uma vídeo aula de Guinther Pauli que apresenta uma introdução a testes unitários utilizando DUnit.

http://www.devmedia.com.br/articles/viewcomp.asp?comp=540

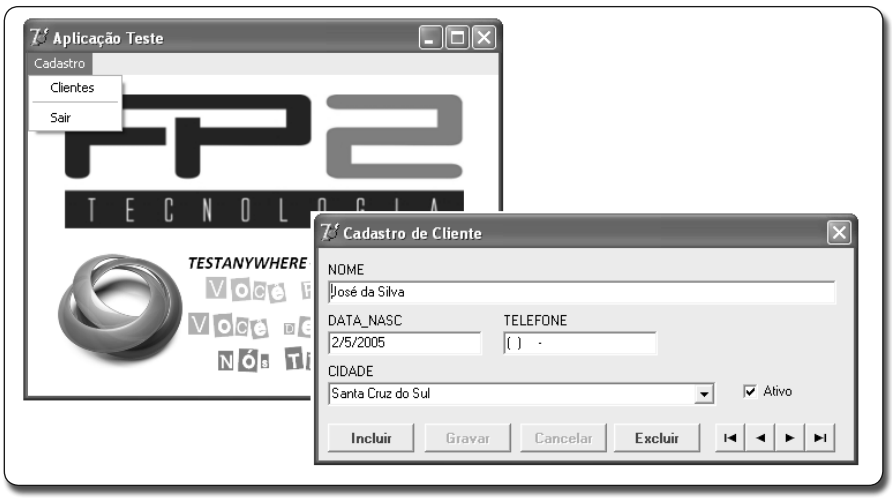


Figura 1. Sistema Demo usada neste artigo

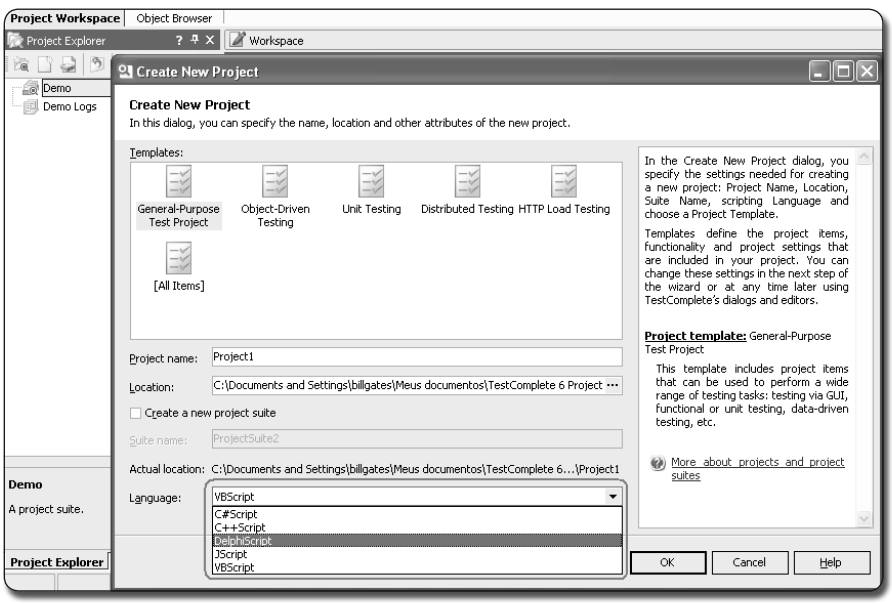


Figura 2. Diálogo de criação de um projeto

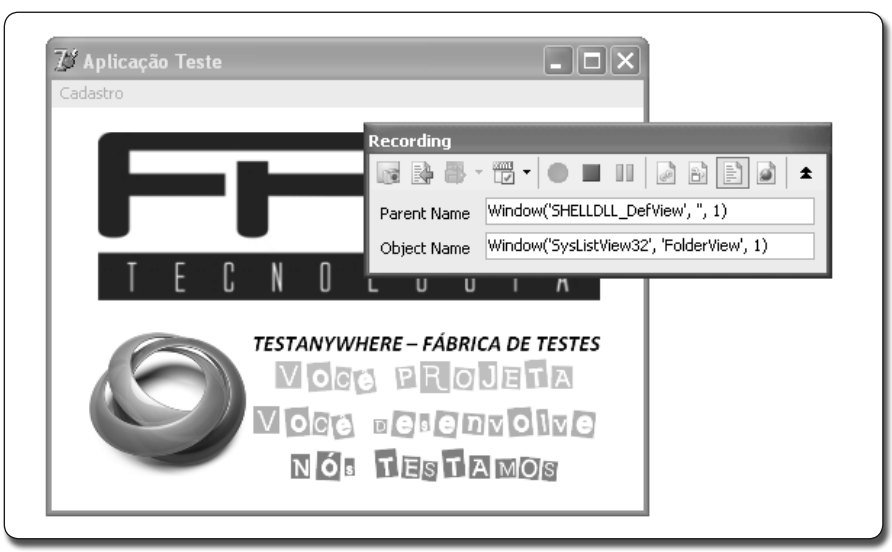


Figura 3. Gravando um script de teste

existe maior probabilidade de existir erros. Independência das mudanças da interface gráfica. Alto desempenho para testes automatizados que exigem centenas de milhares de repetições, testes de funcionalidades que realizam cálculos complexos, integração entre sistemas diferentes e assim por diante.

- *Desvantagens*: Requer grandes modificações no sistema para expor as funcionalidades ao mundo exterior. Exige profissionais especializados em programação para criar os testes automatizados. Existem poucas ferramentas/*frameworks* que suportam essa abordagem (normalmente é necessário criar soluções caseiras).

TestComplete

O *TestComplete* é uma ferramenta de automação de testes de uso comercial. As versões *Enterprise* e *Standard* custam US$ 1,999.00 e US$ 999.00, respectivamente e são desenvolvidas pela empresa *AutomatedQA*. Uma versão de avaliação pode ser encontrada no site da empresa no link *www.automatedqa.com/downloads/testcomplete/index.asp*. Um dos destaques do *TestComplete* é o seu excelente suporte para a automação de testes de sistemas desenvolvidos em Delphi.

A ferramenta oferece recursos para automatizar os testes, transformando as ações dos usuários em um *script* que pode ser reproduzido (*Playback*) posteriormente.

A arquitetura é baseada em *plug-ins*. Estes *plug-ins* são chamados de *Itens de Projeto*. Por exemplo, se você precisar testar um *Web Service*, será necessário adicionar o item de projeto chamado *Web Services* para poder usar essa funcionalidade, e assim por diante. Na **Tabela 1** são apresentados os itens de projeto oferecidos pelo *TestComplete* atualmente (a versão *Enterprise* vem com todos os itens de projeto existentes).

Automação de testes na prática

Nessa parte, apresentaremos as funcionalidades básicas do *TestComplete*. Para fins didáticos, construímos uma aplicação simples no Delphi para demonstrar a utilização da ferramenta em questão. O sistema tem uma tela principal e um

menu que dá acesso a tela de cadastro de clientes com alguns campos básicos, como pode ser observado na **Figura 1**.

Para automatizar os testes é necessário primeiro criar um projeto. Para realizar tal tarefa, devemos abrir o menu *File>New>New Project*. O diálogo de criação de um novo projeto deverá ser exibido e nesse diálogo devemos escolher o *template General-Purpose Test Project*. Observe na **Figura 2**, que é permitido a escolha da linguagem de *script* em que os testes serão criados. Para o nosso exemplo, devemos escolher a opção *DelphiScript*.

Tão logo o projeto seja criado, podemos iniciar a gravação de um *script* de teste automatizado. Para realizar esta tarefa, devemos abrir o menu *Script>Record*. O diálogo *Recording* é exibido (**Figura 3**). A partir desse ponto todas as ações serão gravadas. Em nosso exemplo, vamos abrir a janela de cadastro de cliente por meio do menu *Cadastro>Clientes*, cadastrar um novo cliente, gravar os dados digitados e fechar a janela. Após essas ações devemos clicar no botão *Stop* do diálogo *Recording*.

Quando a gravação é finalizada, o *TestComplete* cria um novo procedimento na *Unit1* e converte todas ações realizadas em instruções na linguagem de *script* alvo, no nosso caso é a *DelphiScript*, como pode ser visto na **Figura 4.**

> **Nota**: O *script* original foi modificado para aumentar a legibilidade desse artigo.

Devemos ressaltar que, durante a gravação, o *TestComplete* não grava os movimentos do mouse, ou cliques nas coordenadas X e Y. O motor (*engine)* de gravação, consegue *entrar* no sistema durante a gravação e identificar os nomes e classes das janelas e objetos, bem como as suas propriedades. Além disso, todas as ações são convertidas em instruções fáceis de entender para quem tem familiaridade com programação, como por exemplo: *ClickButton*, *Click*, *ClickItem*, *Keys*, etc. Dessa forma, a criação e manutenção dos *scripts* de teste fica mais fácil e intuitiva.

Adicionalmente, devemos reforçar que o *TestComplete* oferece um ambiente completo de desenvolvimento para a criação e manutenção dos *scripts* de testes. Dentre os recursos oferecidos, devemos destacar a integração com sistemas de controle de versões e recursos para a depuração (*debug*) dos *scripts* de testes. O recurso de depuração oferece funcionalidades semelhantes ao Delphi, como por exemplo, *breakpoints*, *watch list*, *call stack*, entre outros (**Figura 5**).

Com o código do *script* gravado podemos executar o teste por meio do menu *popup Run current routine* (**Figura 6**). O programa executará todas as ações da mesma forma e ordem em que foram gravadas. Ao final da execução ele exibe um relatório (*Test Log*). O relatório de execução apresenta informações sobre os *scripts* que foram executados, as ações que foram realizadas, os erros ocorridos e demais informações sobre a execução dos testes (**Figura 6**).

Até agora vimos como gravar e executar um *script* de testes. No entanto, não fizemos nenhum teste de verdade, apenas execução automática de ações.

Criando um caso de teste

Um caso de teste é, em princípio, a execução de uma ação no sistema e a avaliação/comparação de um resultado esperado. Para realizar a avaliação do resultado esperado, o *TestComplete* oferece um recurso chamado *Checkpoint*.

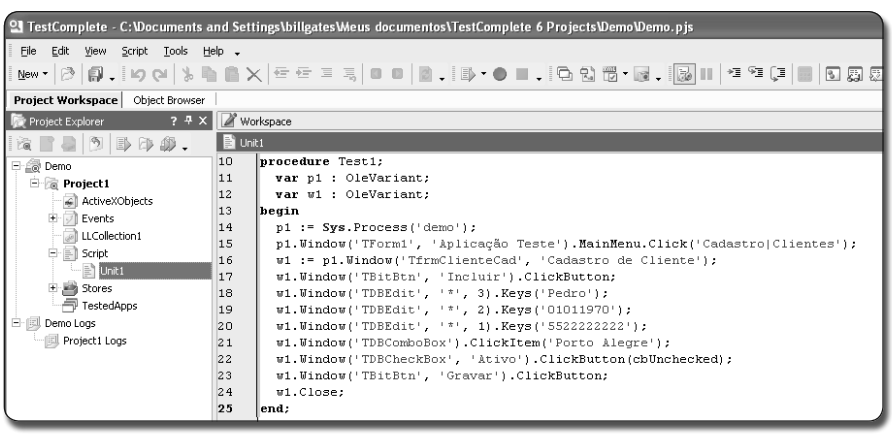


Figura 4. Script de teste criado na linguagem DelphiScript

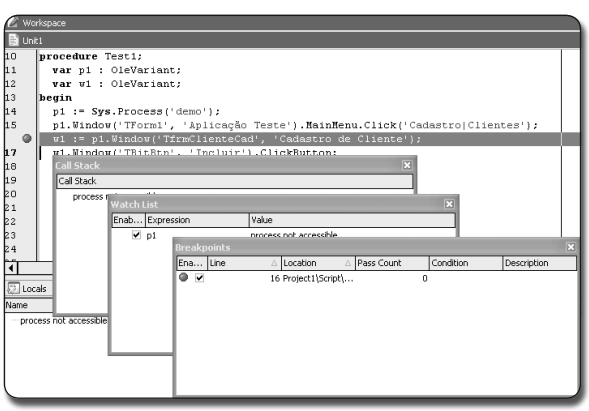


Figura 5. Recursos para depuração de scripts

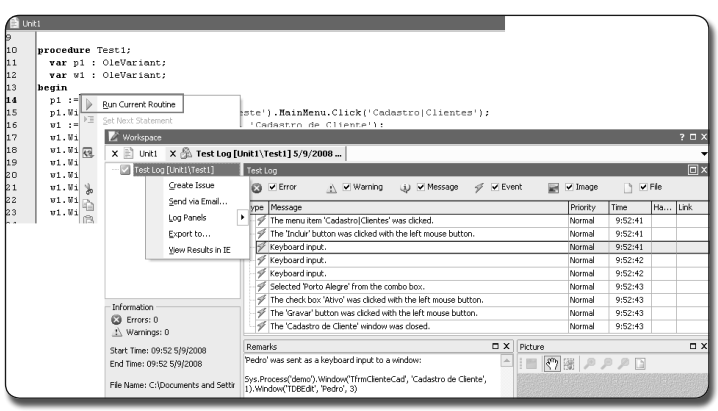


Figura 6. Relatório com o log de execução do script

Um *Checkpoint* é um recurso usado para realizar um ponto de verificação (ou comparação) durante a execução dos testes. Por exemplo, durante a execução de um teste hipotético você usará *Checkpoints* para verificar se o campo *valor total da venda* apresenta o valor esperado, se o botão *Cancelar Venda* está desabilitado e os dados foram inseridos nas tabelas do banco de dados corretamente. Na **Tabela 2** são apresentados os *Checkpoints* suportados pelo *TestComplete*:

Com base no que foi exposto, para criar um *Checkpoint* durante a gravação do teste, devemos clicar no botão *Add Checkpoint from the list* do diálogo *Recording*, como pode ser observado na **Figura 7**. Para fins de análise e entendimento, vamos criar um *Property Checkpoint*. Este *Checkpoint* cria um ponto de verificação onde uma propriedade de um objeto na tela é comparada com um valor pré-definido. Clique na opção *Create Property Checkpoint*. O diálogo *Create Property Checkpoint* deverá ser exibido. Nesse diálogo, devemos escolher o objeto e a propriedade que será comparada com o valor pré-definido. No nosso exemplo o objetivo desse *Checkpoint* é avaliar se o *ComboBox Cidade* apresenta o valor *Porto Alegre* após gravarmos os dados do cadastro. Para isso vamos escolher esse *ComboBox* da janela *Cadastro de Cliente* e a propriedade *wText*. Essa propriedade contém o nome da cidade que é exibido no *ComboBox* (**Figura 8**). Na **Figura 9** podemos ver o *script* gerado.

Checkpoint	Descrição
Property Checkpoints	Permite a criação de pontos de verificação com base nas propriedades dos objetos.
Object Checkpoints	Permite a criação de pontos de verificação de objetos.
Table Checkpoints	Permite a criação de pontos de verificação de *grids* e objetos que apresentem as informações em formato tabular.
File Checkpoints	Permite a criação de pontos de verificação para a comparação de arquivos.
XML Checkpoints	Permite a criação de pontos de verificação para a comparação de arquivos no formato *XML*.
Database Table Checkpoints	Permite a criação de pontos de verificação que realizam a comparação de dados armazenados em tabelas no banco de dados.
Region Checkpoints	Permite a criação de pontos de verificação que realizam a comparação de imagens de regiões da tela.
Web Service Checkpoints	Permite a criação de pontos de verificação dos resultados de uma resposta de um *Web Service*.
Web Accessibility Checkpoints	Permite a criação de pontos de verificação de alguns aspectos de acessibilidade de páginas *WEB*.
Web Comparison Checkpoints	Permite a criação de pontos de verificação para comparar alguns tipos de atributos de páginas *WEB*.

Tabela 2. Tipos de checkpoints

Organizando os Testes

Por fim, após a criação de dezenas ou centenas de *scripts* de testes percebemos que é necessário executá-los numa ordem específica, em grupos ou hierarquicamente. O *TestComplete* oferece um recurso bastante avançado para gerenciar a execução de grupos de *scripts* de testes. Para utilizar esse recurso, você deverá clicar duas vezes sobre o projeto *Project1*.

Nesta janela, selecione a aba *Test Items* e em seguida você poderá incluir os testes que você deseja executar na ordem que quiser, agrupá-los hierarquicamente e configurar o comportamento da execução, tais como: se o teste deverá parar se houver um erro ou exceção, o tempo de espera máximo antes de cancelar a execução de um teste que ficou travado por alguma razão, a quantidade de vezes que um teste deverá ser executado e assim por diante, como pode ser visto na **Figura 10**.

Conclusão

Como observamos neste artigo, a automação de testes não é uma atividade fácil. Além disso, automatizar os testes é uma atividade muito parecida com o desenvolvimento. A automação de testes deve ser encarada como um projeto de desenvolvimento com características e infra-estrutura próprias, ou seja, exige um planejamento detalhado, assim como atividades de projeto e desenvolvimen-

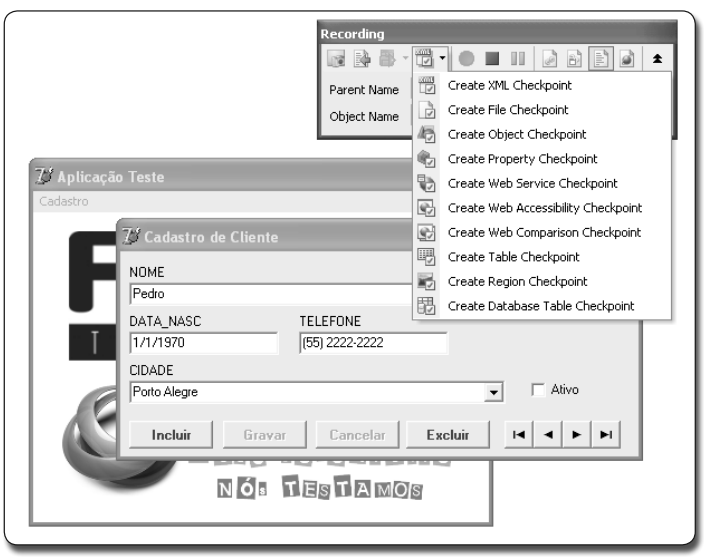


Figura 7. Listagem dos Checkpoints existentes

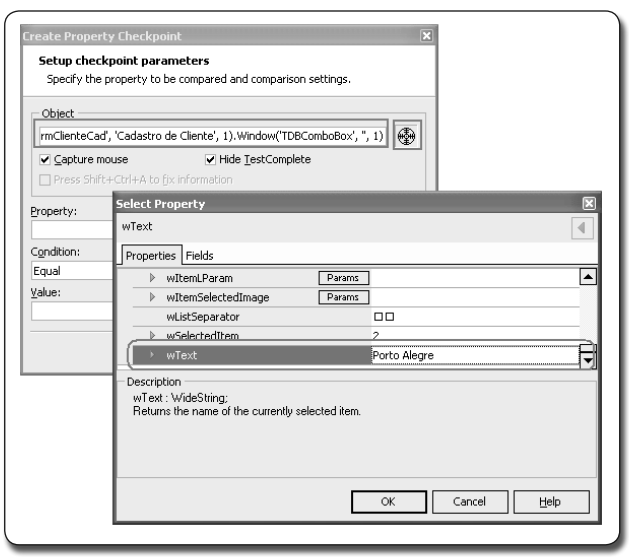


Figura 8. Adicionando um Property Checkpoint

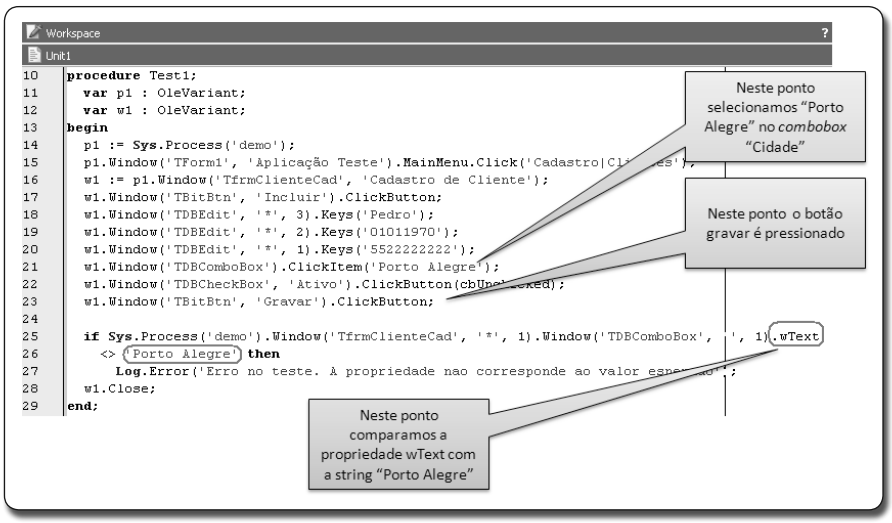


Figura 9. Script de teste contendo um Property Checkpoint

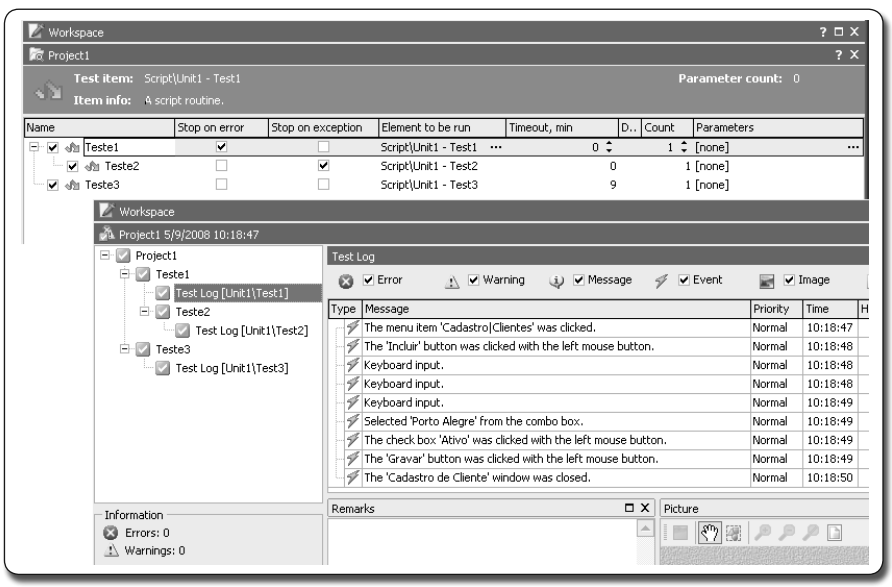
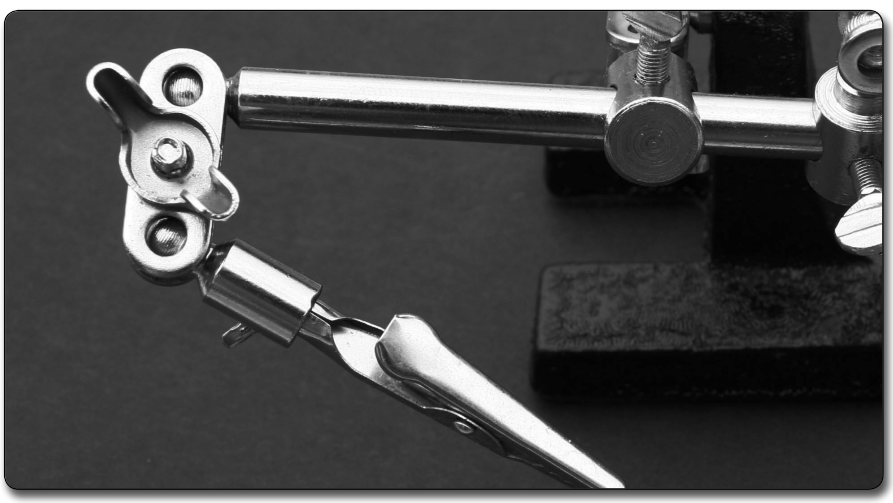


Figura 10. Organização da execução de um conjunto de scripts e o relatório



to/manutenção dos *scripts*, tal qual o desenvolvimento de um software convencional. O investimento em automação de testes não se limita apenas aos custos de aquisição da ferramenta de automação de testes, devemos também considerar a capacitação dos profissionais em desenvolvimento de software.

Segundo Cem Kaner, autor do livro *Lessons Learned in Software Testing*, o propósito da automação de testes pode ser resumidamente descrito como a aplicação de estratégias e ferramentas tendo em vista a redução do envolvimento humano em atividades manuais repetitivas. Dessa forma, devemos reconhecer que a automação de testes não deve ser empregada como um substituto do teste manual. Ela deve ser introduzida como uma técnica adicional cujo objetivo principal é agregar valor, sem, no entanto, invalidar o teste manual existente. Afinal, testes manuais e automatizados são abordagens de testes diferentes que se reforçam mutuamente. ●

Links

TestAnywhere – TestComplete no Brasil
www.testanywhere.com.br

AutomatedQA – Fabricante do TestComplete
www.automatedqa.com

TestExpert – Comunidade Brasileira de Teste de Software
www.testexpert.com.br

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Nesta seção você encontra artigos sobre técnicas que poderão aumentar a qualidade do desenvolvimento de software

Suporte técnico on-line via chat

Faça você mesmo um aplicativo de Chat usando Delphi

Resumo DevMan

Manter uma forma de comunicação barata e eficiente com o cliente é essencial. E se esse meio de comunicação puder ser personalizado para sua empresa, melhor ainda. Aumentar a comunicação do cliente com o suporte técnico e ter essa preocupação com o bom atendimento, sem dúvida é o diferencial de qualquer empresa. Hoje o mercado tem usado freqüentemente o Messenger da Microsoft para manter um bom relacionamento com seus clientes, porém muitas empresa temem o uso indiscriminado da ferramenta, além é claro do risco de vírus na rede.

Neste artigo um serviço de troca de mensagens instantâneas será desenvolvido e personalizado de acordo com as nossas necessidades. Criaremos um pequeno protocolo para troca de mensagens e um servidor para distribuir o fluxo de comandos.

Nesse artigo veremos

- Como utilizar a suíte de componentes Indy;
- Como criar um protocolo de comunicação;
- Como utilizar o protocolo criado em um aplicativo de troca de mensagens instantâneas.

Qual a finalidade

- Desenvolvimento de um servidor de mensagens no estilo MSN.

Quais situações utilizam esses recursos?

Em sistemas que necessitam trocar informações através de uma rede TCP/IP.

Paulo Quicoli

(pauloquicoli@devmedia.com.br)

é Editor Técnico da revista ClubeDelphi, formado em processamento de dados pela FATEC-TQ. Atua como analista e desenvolvedor da ControlM Informática (www.controlm.com.br). Utiliza Delphi desde de sua primeira versão, entusiasta do desenvolvimento orientado a objetos tem publicado vários artigos sobre o assunto. Blog: www.pauloquicoli.spaces.live.com

Acredito que a uma boa parte de nós, programadores que precisam manter contato com seus clientes, utilizam o *MSN* para prestar suporte técnico. Porém existem alguns clientes que não permitem o uso de aplicativos de mensagem instantânea com medo de que sua rede seja infectada por vírus ou até mesmo por achar que os funcionários ficarão batendo papo o dia todo. Como resolver esse impasse? É aí que o Delphi vem em nosso socorro. Por que não criar um serviço de *chat*, estilo *MSN*, só que personalizado para sua empresa? O cliente com esse *chat* instalado não veria outra coisa a não ser a disponibilidade do suporte e você poderia ver todos os clientes que estão *on-line*. Com certeza ao mostrar esse aplicativo a esse cliente ele iria se sentir seguro, pois o serviço de comunicação estará com o logo da sua empresa e apenas o setor de suporte estará disponível. Vamos por a mão na massa e agregar valor aos nossos negócios!

Uma visão geral

Nosso *DevChat*, nome que daremos a nossa aplicação, será composto de 3 partes: um *server* e dois *clients*. O *server* será responsável por manter os *clients* cadastrados e seu estado. Também caberá a ele direcionar a comunicação entre os *clients*. Isso significa que para um *client* enviar uma mensagem de texto para outro, essa deverá ser enviada ao *server* e então será encaminhada para o destinatário (**Figura 1**) e para isso o *client* deve se conectar ao *Server* por meio da Internet.

Os outros dois *clients* são muito parecidos. Um será destinado aos clientes e a ele só será permitido enxergar a equipe de suporte. O outro será utilizado pelo suporte e nele estarão sendo exibidos todos os clientes.

Mas o que esperar do nosso *DevChat* ? Quero implementar junto com vocês as seguintes funcionalidades:

• *Troca de mensagens, formato texto simples*: Quem nunca sentiu dificuldade em ler certos diálogos no *MSN* onde a cada palavra que a pessoa digita do outro lado forma um *gif* animado pra você? Nosso *chat* será uma ferramenta de suporte da sua empresa, por isso, isso será cortado.

• *Notificação de mudança de estado:* Será possível identificar quem está on-line ou não.

Acredito que com esses requisitos estamos cobrindo o que há de mais básico em termos de comunicação textual *on-line*.

Construindo o servidor

Como já mencionado, o *server* deve manter os *clients* cadastrados. Para isso vou utilizar um banco de dados *Firebird*. Na **Figura 2** vemos as tabelas que o compõem. Além disso, também é sua responsabilidade validar quem está tentando se conectar. Para que o exemplo faça sentido, crie um banco de dados com o nome que desejar, aqui criamos com o nome *DevChat.fdb*, e nele crie as tabelas, chaves primárias e estrangeiras conforme o esquema da **Figura 2**. Não abordarei a criação do banco nesse artigo por não fazer parte do tema.

> **Nota:** Não cobrirei aqui a forma que esses *clients* serão cadastrados, por isso vou inserir alguns registros no banco de dados de forma manual. Seria importante disponibilizar, por exemplo, uma página onde seus clientes pudessem baixar o *client* e registrar um usuário e senha.

A comunicação realizada em um aplicativo de *Chat* se dá através de uma rede *TCP/IP*, ou seja, para criarmos nosso servidor ele deve ser capaz de utilizar *TCP/IP* para enviar e receber mensagens. Como fazer isso com *Delphi*? Quando você instala o *Delphi* um dos pacotes de componentes que é adicionado é o *Indy*. Ele é composto por um conjunto de componentes específicos para trabalhar com redes e oferece os mais variados recursos. Neste artigo vamos utilizar a versão disponibilizada junto com o *RAD Studio 2007*. Portanto crie um novo projeto no

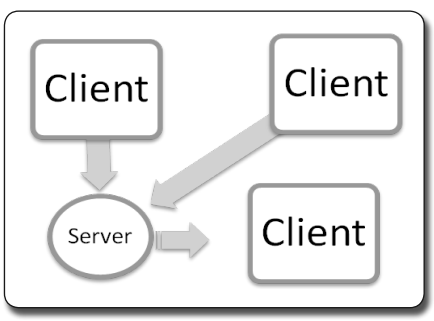


Figura 1. Interação entre clients e server

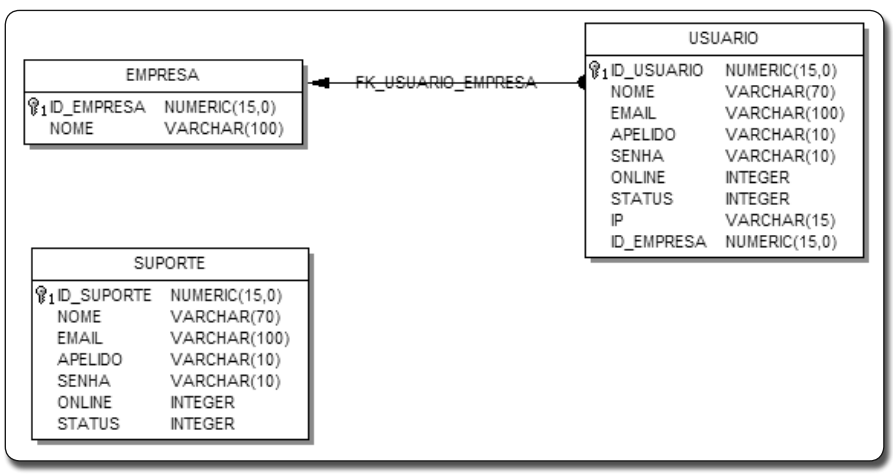


Figura 2. Diagrama do banco de dados DevChat.fdb

Figura 3. Exemplo de tela do DevChat-Server

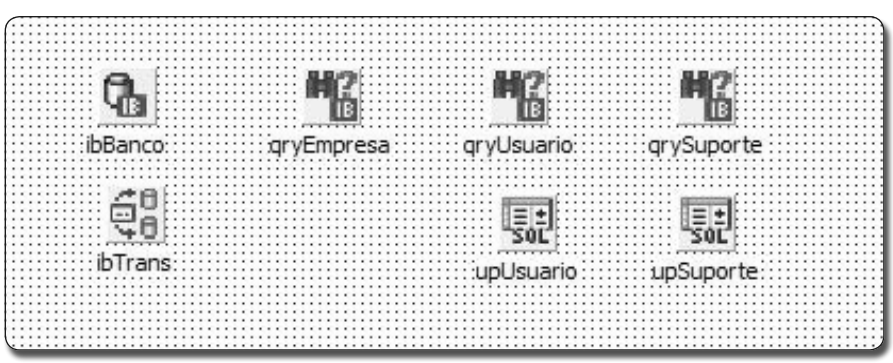


Figura 4. Data Module para conexão ao banco

RAD Studio e salve-o como *DevChatServer.dproj* e deixe-o como na **Figura 3.**

Adicione também um *Data Module* ao projeto e conecte-se ao banco de dados, como na **Figura 4**. Os componentes *qryEmpresa, qryUsuario, qrySuporte* simplesmente acessam suas respectivas tabelas e juntamente com os controles de *update* (*upUsuario* e *upSuporte*) permitem que sejam atualizadas informações.

Nota: Para conexão com o banco de dados, estamos usando os componentes na paleta *Interbase*, tais como *IBDataBase, IBQuery, IBTransation* e *IBUpdateSQL*. Porém, nada impede que façamos tais conexões utilizando outros componentes de acesso a dados como o *dbExpress*, por exemplo.

Ao formulário principal adicione um componente *idTCPServer* que está localizado na guia *Indy Servers*. É este componente que pode tornar nosso aplicativo um servidor. Nele configuramos qual porta o *DevChat* irá utilizar para *escutar* os *clients* que desejam se conectar. Para ativar o servidor vamos colocar no evento *OnClick* do botão *Iniciar* o código da **Listagem 1**.

Se executarmos o programa e clicarmos sobre o botão *Iniciar* é provável que o *firewall* do seu sistema operacional seja acionado. Isso acontece porque estamos abrindo uma porta de conexão. No caso do Windows XP, basta confirmar e tudo estará funcionando. Caso o exista algum *firewall* no seu sistema operacional que não seja o do próprio Windows, talvez seja necessário dar permissão para o sistema que acabamos de desenvolver.

Na **Listagem 1** estamos passando, através da propriedade *DefaultPort*, para o componente *tcpServer* qual porta será utilizada para manter uma comunicação e então o ativamos. Após isso, ajustamos todos os usuários a terem seu status como *off-line* apenas fazendo um *Update* nas tabelas *Usuario* e *Suporte*. Isso é necessário, já que em teoria nenhum usuário ou suporte técnico poderia estar *on-line* com o servidor *off-line*.

Como dito anteriormente, o servidor ficará escutado as requisições das outras duas aplicações e fará o redirecionamento das mensagens de um para outro. Por isso, precisaremos prever que tipos de mensagem e como elas serão interpretadas no lado servidor.

O que precisamos fazer agora é programar o servidor para que ele seja capaz de interpretar o que recebe e conseqüentemente distribuir as mensagens e comandos recebidos. Usaremos o evento *OnExecute* do componente *TIdTCPServer*. Clique duas vezes sobre esse conforme visto na **Listagem 2**.

Nessa **Listagem 2**, recebemos o valor passado pelo *client* através do parâmetro do evento. Em seguida fazemos um *copy* na variável *Texto* para que possamos saber qual *comando* ou mensagem foi passado. Caso a mensagem seja *listaSuporte* chamamos o método *GetUsuariosSuporte*. Você precisará criar essa função conforme a **Listagem 3**. Veja que a tarefa é simples, apenas conectamos ao banco

Listagem 1. Iniciando o servidor

```
procedure TfrmPrincipal.btnIniciarClick(Sender: TObject);
begin
  if btIniciar.Caption = 'Iniciar' then
  begin
    tcpServer.DefaultPort := strtoInt(editPorta.Text);
    tcpServer.Active := true;
    dtm.qryUsuario.Close;
    dtm.qryUsuario.SQL.Clear;
    dtm.qryUsuario.SQL.Text := 'UPDATE USUARIO SET ONLINE = 0, STATUS = 0';
    dtm.qryUsuario.ExecSQL;
    dtm.qryUsuario.Transaction.CommitRetaining;
    dtm.qryUsuario.Close;
    dtm.qrySuporte.Close;
    dtm.qrySuporte.SQL.Clear;
    dtm.qrySuporte.SQL.Text := 'UPDATE SUPORTE SET ONLINE = 0, STATUS = 0';
    dtm.qrySuporte.ExecSQL;
    dtm.qrySuporte.Transaction.CommitRetaining;
    dtm.qrySuporte.Close;
    btIniciar.Caption := 'Parar';
  end
  else
  begin
    tcpServer.Active := false;
    btIniciar.Caption := 'Iniciar'
  end;
  Total := 0;
end;
```

Listagem 2. Processando o comando de lista de suporte

```
procedure TServerF.tcpServerExecute(AContext: TIdContext);
var
  comando: string;
  Texto: string;
  ListaSuporte: TStringList;
begin
  Texto:= AContext.Connection.IOHandler.ReadLn;
  comando := Copy(Texto, 1, Pos('||', Texto)-1);
  if comando = 'listaSuporte' then
  begin
    ListaSuporte := GetUsuariosSuporte;
    AContext.Connection.IOHandler.WriteRFCStrings(ListaSuporte);
  end;
end;
```

Listagem 3. Obtendo a lista de suporte

```
function TServerF.GetUsuariosSuporte: TStringList;
var
 ListaSuporte: TStringList;
begin
  dtm.qrySuporte.Close;
  dtm.qrySuporte.SQL.Clear;
  dtm.qrySuporte.SQL.Add('SELECT * FROM SUPORTE');
  dtm.qrySuporte.Open;
  ListaSuporte := TStringList.Create;
  while not dtm.qrySuporte.Eof do
  begin
    ListaSuporte.Append('listaSuporte||' +
      dtm.qrySuporteID_SUPORTE.asString + '||' +
      dtm.qrySuporteAPELIDO.AsString + '||'+
      dtm.qrySuporteONLINE.AsString + '||'+
      dtm.qrySuporteSTATUS.AsString+ '||');
    dtm.qrySuporte.Next;
  end;
  dtm.qrySuporte.Close;
  result := ListaSuporte;
end;
```

de dados, executamos uma instrução SQL para retornar todos os membros do suporte técnico e carregamos em um *TStringList* que é passado como retorno.

Mais tarde veremos que a lista retornada será utilizada para armazenar esses integrantes do suporte na versão *client* e para isso vamos utilizar o que o *Delphi* nos oferece de melhor, um *ClientDataSet*.

Mas antes disso veremos outro ponto importantíssimo em nossa solução. Para que as três aplicações conversem entre si, vamos estabelecer um protocolo. Definiremos algumas regras e comandos que farão com que os três aplicativos enviem e recebam mensagens e comandos entre si.

No servidor é preciso agora verificar se o usuário que está tentando se conectar é válido. Toda vez que algum usuário tenta se conectar utilizando o *host* e porta específicos, o evento *onConnect* do *TcpServer* é disparado e é nele que vamos verificar as informações que estão chegando. Programe o evento *OnConnect* mencionado anteriormente conforme a **Listagem 4.**

Vamos entender a **Listagem 4**. Observe que o evento *tcpServerConnect* carrega consigo um parâmetro do tipo *TIdContext*, o *AContext*. Para cada *client* que se conecta, ou tenta se conectar, um novo *TIdContext* é criado o representando. A variável *AContext* contém informações importantes, como por exemplo a conexão atual, e através dela podemos obter os comandos que o *client* está enviando. Isso é feito na linha a seguir:

```
texto :=
  AContext.Connection.IOHandler.ReadLn;
```

Perceba que em dado momento, recebemos o *ID* do usuário na variável *idUsuario*. Essa variável recebe a identificação do usuário através da função *Login*.

```
idUsuario := Login(
    loginInfo.IdEmpresa,
      loginInfo.Nick, loginInfo.Senha);
```

Veja que desmembramos o conteúdo da variável texto em um objeto da classe *TLoginInfo* e passamos o resultado para o método *Login*, que por sua vez retorna o *Id* do usuário, 0 se o *login* não for válido ou ainda -1 se o usuário já estiver conectado. Caso o usuário seja válido, um objeto do tipo *TUsuario* é instanciado e passado para a propriedade *Data* do objeto *AContext* atual, então enviamos o retorno disso de volta para o *client*, que digo novamente é representado pelo objeto *AContext*. Atualizamos a quantidade de usuários conectados no *server*.

Nota: A definição da classe *TLoginInfo* e *TUsuario* é feita na *Unit ClientesU* e por questões de espaço não é exibida aqui mas está disponível para *download*.

Crie uma nova função no *Server* para que possamos fazer esse trabalho. Veja seu código na **Listagem 5**. Na **Listagem 5**

Listagem 4. Validando a conexão

```
procedure TServerF.tcpServerConnect(AContext: TIdContext);
var
  cliente: TUsuario;
  loginInfo: TLoginInfo;
  IdUsuario: integer;
  texto, comando: string;
begin
  texto := AContext.Connection.IOHandler.ReadLn;
  comando := Copy(texto,1,pos('||',texto)-1);
  delete(texto,1,pos('||',texto)+1);
  loginInfo := TLoginInfo.Create;
  loginInfo.IdEmpresa := strToInt(Copy(texto,1,pos('||',texto)-1));
  delete(texto,1,pos('||',texto)+1);
  loginInfo.Nick := Copy(texto,1,pos('||',texto)-1);
  delete(texto,1,pos('||',texto)+1);
  loginInfo.Senha := Copy(texto,1,pos('||',texto)-1);
  idUsuario := Login(loginInfo.IdEmpresa, loginInfo.Nick, loginInfo.Senha);
  if idUsuario > 0  then
  begin
    cliente := TUsuario.Create;
    cliente := LoadUsuario(idUsuario);
    cliente.IP := AContext.Connection.Socket.Binding.PeerIP;
    cliente.Host := GStack.HostByAddress(cliente.IP);
    cliente.Online := oSim;
    cliente.Status := sDisponivel;
    AContext.Data := cliente;
    AContext.Connection.IOHandler.WriteLn('sucessoLogin||'
      + IntToStr(cliente.IdUsuario)+ '||');
   Total := Total + 1;
  end
  else
  begin
    AContext.Connection.IOHandler.WriteLn('sucessoLogin||'
      + IntToStr(idUsuario)+'||');
    Acontext.Connection.Disconnect;
  end;
  loginInfo.Free;
end;
```

Listagem 5. Verificando o usuário junto ao banco de dados

```
function TServerF.Login(IdEmpresa: integer; Nick, Senha: string): Integer;
begin
  dtm.qryUsuario.Close;
  dtm.qryUsuario.SQL.Clear;
  dtm.qryUsuario.SQL.Add('SELECT * FROM USUARIO');
  dtm.qryUsuario.SQL.Add('WHERE ID_EMPRESA = :ID_EMPRESA');
  dtm.qryUsuario.SQL.Add('AND APELIDO = :NICK');
  dtm.qryUsuario.SQL.Add('AND SENHA = :SENHA');
  dtm.qryUsuario.ParamByName('NICK').AsString := Nick;
  dtm.qryUsuario.ParamByName('ID_EMPRESA').AsInteger := IdEmpresa;
  dtm.qryUsuario.ParamByName('SENHA').AsString := Senha;
  dtm.qryUsuario.Open;
  if dtm.qryUsuario.IsEmpty then
    result := 0
  else
  begin
    if dtm.qryUsuarioONLINE.AsInteger = 1 then
      result := -1
    else
    begin
      result := dtm.qryUsuarioID_USUARIO.AsInteger;
      dtm.qryUsuario.Edit;
      dtm.qryUsuarioONLINE.AsInteger := 1;
      dtm.qryUsuarioSTATUS.AsInteger := 1;
      dtm.qryUsuario.Post;
      dtm.qryUsuario.Transaction.CommitRetaining;
    end;
  end;
  dtm.qryUsuario.Close;
end;
```

Funcionalidade	Protocolo
Login	login||IdEmpresa||<Nick>||senha||
Obter lista dos usuários de suporte	listaSuporte||IdUsuario||
Enviar mensagem para alguém do suporte	mensagemParaSuporte||IdUsuarioOrigem||IdSuporte||mensagem||NumeroLinha
Estou escrevendo	Escrevendo||IdUsuarioOrigem||IdUsuarioDestino||
Parei de escrever	pareiEscrever||IdUsuarioOrigem||IdUsuarioDestino||
Mudança de estado	mudancaEstado||IdUsuario||Estado||OnLine||
Obter a lista dos clientes cadastrados e seus status	listaCliente||<IdSuporte>||
Login do client de suporte	loginSuporte||<Nick>||senha||
Mensagem para algum cliente	mensagemParaCliente||IdSuporteOrigem||IdCliente||mensagem||NumeroLinha

Tabela 1. Comandos do protocolo

Funcionalidade	Protocolo
Retornar a lista de suporte	Um stringlist onde cada item obedece: listaSuporte||idSuporte||Nick||OnLine||estado||
Sucesso/Falha de conexão/login	sucessoLogin||IdCliente||

Tabela 2. Comandos de retorno do server

vemos como que um usuário é validado. Criamos uma *function* chamada *Login* que é chamada de dentro do evento *OnConnect* retornando o *ID* do usuário. A validação é ainda mais simples do que vimos até agora. Apenas pegamos a *Empresa, Apelido* e *Senha* do usuário e efetuamos consultas ao banco de dados.

Definindo um protocolo

Como o *client* precisa se comunicar com o servidor é necessário que criemos um protocolo de comunicação que seja obedecido e conhecido pelas partes envolvidas. Um protocolo de comunicação é um conjunto de comandos e valores que são formatados em um padrão pré-determinado. Nosso protocolo será simples, baseado em texto simples e cobrirá as funcionalidades citadas anteriormente. Veja que nas **Tabelas 1** e **2** especificamos os o comandos válidos que serão trocados entre aplicação cliente e servidora, respectivamente.

Basicamente nosso protocolo é composto por um comando e seus parâmetros, todos em um texto. Para que possamos distinguir o que é cada valor, criamos um separador para nos ajudar. Nosso separador é composto por dois *pipe-lines*, ||, colocados em seqüência. Para que possamos ver o protocolo em funcionamento precisamos implementar tanto o *client* quanto o *server* em paralelo. Já vimos um exemplo de uso do protocolo na **Listagem 2.** Veja:

```
if comando = 'listaSuporte' then
begin
  ListaSuporte := GetUsuariosSuporte;
  AContext.Connection.IOHandler.
    WriteRFCStrings(ListaSuporte);
end;
```

Aqui estamos verificando se o comando recebido é a palavra *listaSuporte* e caso seja, carregamos os usuários como já vimos anteriormente.

Criando a aplicação cliente

Teremos dois *clients*. Um será utilizado por nossos clientes e outro pela equipe de suporte que terá funcionalidades especiais como veremos mais adiante. Vamos começar pelo dos clientes. Adicione à solução um novo projeto, do tipo *Win32*, chamando-a de *DevChatClient.dproj*. No formulário principal vamos adicionar um *TIdTCPClient* da guia *Indy Clients* e configurar suas propriedades *Port* e *Host* para que sejam as mesmas utilizadas pelo servidor de mensagens. Em *Host* colocamos o *IP* da máquina onde está o servidor de mensagens, que neste caso é *127.0.0.1* ou *localhost*. Adicione agora um painel e nele insira três *TEdit* e um *TButton*, como na **Figura 5**.

No *OnClick* do botão tentamos conectar ao servidor de mensagens, como é mostrado na **Listagem 6**.

Perceba que o código não é em nada complexo. Estamos verificando se os campos *Empresa, Login* e *Senha* estão preenchidos e em caso positivo chamamos o método *Connect* do componente *TCPClient*. Esse método se encarregará de conectar-se à aplicação *server* que deverá estar em execução. Após isso, enviamos a mensagem *login||NomeDaEmpresa||Login||Senha||* (*Veja a utilização do nosso protocolo novamente*). Essas informações fazem parte do protocolo que criamos, ou seja, nada mais são do que um simples texto que será recebido na aplicação *Server* e conseqüentemente validado.

Depois de conectado ao servidor, nossa aplicação deverá listar os membros da equipe de suporte e verificar o status de cada um. Lembre-se qie para isso programamos no *Server* uma função capaz de nos retornar tal informação. O retorno da função é um *TStringList*, ou seja, uma lista de *strings*. Como já foi dito, usaremos um *ClientDataset* para preencher com esse membros.

Adicione então um *ClientDataSet* no formulário principal do *Client* e adicione os campos que representam um usuário do suporte: *IdSuporte, Nick, OnLine e Status*. Adicione também um *DataSource* e ligue-o ao *ClientDataSet*. Agora adicione um *DBGrid*, pois é nele que exibiremos a lista de suporte.

Para que o *client* processe o retorno é necessário criar duas classes como na **Listagem 7**, para que além de tratar as mensagens recebidas a resposta do *client* seja mantida.

Basicamente essas duas classes trabalham juntas na leitura de qualquer mensagem enviada pelo servidor. Elas asseguram que a *thread* principal da VCL não será utilizada quando as mensagens chegam, dessa forma a interface gráfica fica livre para que o usuário a utilize. A classe *TReadingThread* lê a mensagem primeiro e então utiliza a classe *TLog* para processá-la. Antes de podermos utilizar as classes *TLog* e *TReadingThread*, vamos implementá-las como na **Listagem 7**.

Crie essas duas classes na *Unit* do formulário principal da versão *client*. Basta digitar suas declarações abaixo da palavra reservada *Type* e em seguida pressionar *Ctrl + Shift + C* para que o

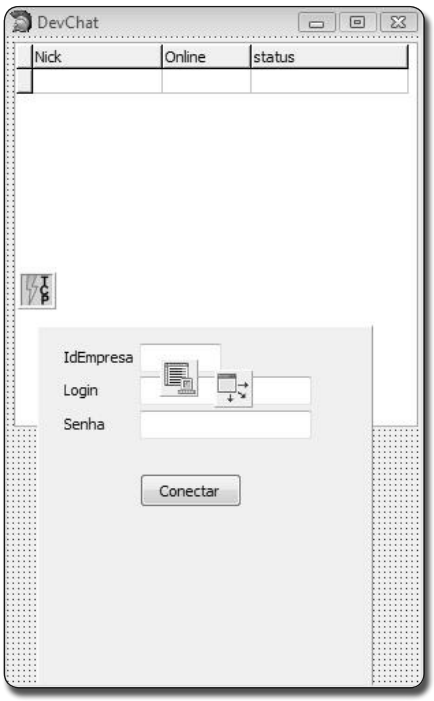


Figura 5. Tela inicial do Client

Listagem 6. Conectando no servidor

```
procedure TClientF.btConectarClick(Sender: TObject);
begin
  if (Trim(EditEmpresa.Text) <> '') and
    (Trim(EditLogin.Text) <> '') and
    (Trim(EditSenha.Text) <> '') then
  begin
    tcClient.Connect;
    tcClient.Socket.WriteLn('login||'+ EditEmpresa.Text +
      '||' + EditLogin.Text + '||'+ EditSenha.Text+'||');
  end
  else
  begin
    ShowMessage('Preencha todas as informações para Login');
    EditEmpresa.SetFocus;
  end;
end;
```

Listagem 7. Classes que mantém a responsividade

```
  TReadingThread = class(TThread)
    protected
      FConn: TIdTCPConnection;
      procedure Execute; override;
    public
      constructor Create(AConn: TIdTCPConnection); reintroduce;
  end;

  TLog = class(TIdSync)
    protected
      FMsg: string;
      procedure DoSynchronize; override;
    public
      constructor Create(const AMsg: string);
      class procedure AddMsg(const AMsg: string);
  end;
```

ClubeDelphi PLUS www.devmedia.com.br/clubedelphi/portal.asp

Acesse agora o mesmo o portal do assinante ClubeDelphi e assista a uma vídeo aula de Guinther Paulo que mostra como trabalhar com thread.

http://www.devmedia.com.br/articles/viewcomp.asp?comp=1733

Delphi nos crie o escopo principal de cada método. Um método *DoSynchronize* fará o papel de sincronizar as mensagens entre as aplicações independente do que ocorrendo no restante do aplicativo.

Precisamos também adicionar na *procedure Synchronize* o tratamento do retorno da lista. Parte do código dessa *procedure* pode ser visto na **Listagem 8**. Evidentemente a listagem não foi totalmente impressa no artigo devido ao seu tamanho. É nesse procedimento que prevemos e tratamos todas as mensagens enviadas e recebidas do servidor. Por exemplo: quando o nosso suporte técnico enviar uma mensagem ao cliente, essa mensagem será recebida com a *string mensagemParaCliente*. Encontre no trecho de código da **Listagem 8** esse pedaço do código e veja o que fazemos.

Basicamente, apagamos os *pipe-lines* de nosso protocolo, pegamos o código do suporte, a mensagem e o número dela e fazemos o *start* da conversa chamando o método *IniciaChatCom*.

Nota do DevMan

O que são Thread?

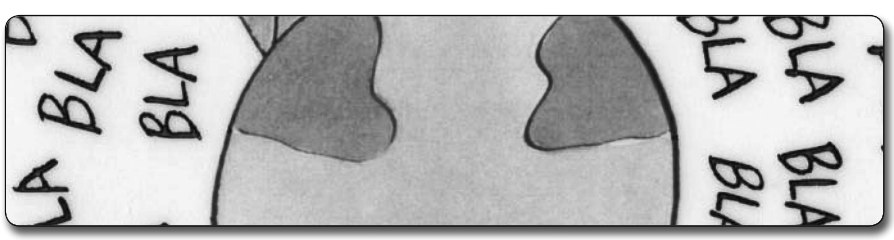
Um programa normal é chamado de *single thread*, ou seja, processo simples ou processo único. Isso significa que só há processamento em um determinado ponto do programa como a emissão de um relatório, cadastramento de um cliente ou mesmo manipulação de qualquer tipo de informação.

Uma *thread* é um processo que acontece com início, meio e fim, e também acontece em um determinado momento dentro de um sistema. Porém, quando temos mais de um processo dentro do mesmo programa chamamos isso de *multithread*. *Multithread* porque temos mais de um processo acontecendo ao mesmo tempo, como emissão de um relatório e cadastramento de um cliente ao mesmo tempo.

As threads são usadas normalmente para agilizar processos ou criar vários processos iguais, porém trabalhando em momentos diferentes. Por exemplo: podemos solicitar a emissão de um relatório grande que levará 2 horas para ser emitido pelo programa devido a massa de dados que ele irá manipular. Se desenvolvermos um relatório como esse em nosso sistema e o usuário utilizá-lo, certamente serão 2 horas sem poder utilizar o programa caso seja *single thread*.

A partir de momento que o isolamos em uma thread, temos condições de liberar o programa para que ele execute novas tarefas sem precisar aguardar o término da primeira.

Um exemplo bastante simples de se entender é quando estamos utilizando programas especializados em downloads de arquivos, como o antigo *Kazaa* ou o mais recente *LimeWire* para download de músicas. Imagine se cada música que você ordenasse o programa a baixar você tivesse que aguardar a última música ser totalmente baixada? Levaria muito tempo. Portanto, esses programas costumam ser *multithread*, pois executam mais de um processo ao mesmo tempo de forma independente.



Listagem 8. Procedure Synchronize no Client

```
procedure TLog.DoSynchronize;
var
  comando: string;
  RetornoLogin: integer;
  CodigoSuporte: integer;
  NickSuporte: string;
  OnLineSuporte: integer;
  StatusSuporte: integer;
  Mensagem: string;
  NumeroLinha: integer;
begin
  comando := Copy(FMsg,1,pos('||',FMsg)-1);
  if comando = 'sucessoLogin' then
  begin
    delete(FMsg,1,pos('||',FMsg)+1);
    RetornoLogin:= strToInt(Copy(FMsg,1,pos('||',FMsg)-1));
    case RetornoLogin of
      -1 : Form1.JaEstavaConectado;
       0 : Form1.LoginDeuErrado;
       else
       begin
         Form1.IdCliente := RetornoLogin;
         Form1.LoginDeuCerto;
       end;
    end;
    exit;
  end;
  if comando = 'listaSuporte' then
  begin
    delete(FMsg,1,pos('||',FMsg)+1);
    CodigoSuporte := strToInt(Copy(FMsg,1,pos('||',FMsg)-1));
    delete(FMsg,1,pos('||',FMsg)+1);
    NickSuporte := Copy(FMsg,1,pos('||',FMsg)-1);
    delete(FMsg,1,pos('||',FMsg)+1);
    OnLineSuporte := strToInt(Copy(FMsg,1,pos('||',FMsg)-1));
    delete(FMsg,1,pos('||',FMsg)+1);
    StatusSuporte := strToInt(Copy(FMsg,1,pos('||',FMsg)-1));
    Form1.AtualizaListaSuporte(CodigoSuporte, NickSuporte, OnLineSuporte,
StatusSuporte);
  end;
  if comando = 'mensagemParaCliente' then
  begin
    delete(FMsg, 1, Pos('||', FMsg)+1);
    CodigoSuporte := strToInt(Copy(FMsg,1,pos('||',FMsg)-1));
    delete(FMsg,1,pos('||',FMsg)+1) ;
    delete(FMsg,1,pos('||',FMsg)+1) ;
    Mensagem := Copy(FMsg,1,pos('||',FMsg)-1);
    delete(FMsg,1,pos('||',FMsg)+1);
    NumeroLinha := strToInt(Copy(FMsg,1,pos('||',FMsg)-1));
    Form1.IniciaChatCom(CodigoSuporte,Mensagem, NumeroLinha);
  end;
[...]
end;
```

Nota: A implementação da função *IniciaChatCom* pode ser vista no arquivo de download disponível no site de edição no portal DevMedia.

Toda a *procedure DoSynchronize* funciona dessa forma. O mais interessante é que podemos implementar diversas funcionalidades somente alterando nosso protocolo e prevendo as mensagens em ambas aplicações, *Server* e *Client*.

A classe *TReadingThread* possui apenas dois métodos: *Create* e *Execute*. No *Create* apenas inicializamos a variável *FConn* e herdamos o método *Create* de seu ancestral. Já o método *Execute*, esse é encarregado de *escutar* a aplicação e detectar o exato momento de uma conexão. Caso uma conexão seja interceptada, então uma mensagem é gerada e enviada para *procedure AddMsg* da classe *TLog*. Isso é necessário para que possamos ter controle de tudo que é feito no sistema. O código de implementação dos métodos das classes *TReadingThread* e *TLog* podem ser vistos na **Listagem 9**.

Note que o método *AddMsg* cria uma instância da classe e sincroniza. O construtor *Create* apenas atualiza a variável *FMsg* e herda o método *Create* de seu ancestor. Por fim o método *DoSynchronize*, que é responsável por fazer o sincronismo das mensagens funcionar perfeitamente. Recebmos o comando e avaliamos. Em caso de sucesso, apagamos os caracteres "||" (*pipe-line*) e testamos o retorno que pode ser 0 ou 1.

Quero destacar que a classe *TLog* é o núcleo do *client*, visto que ela é quem irá processar as mensagens recebidas e irá tomar as devidas providências para que a solicitação/resposta seja feita de acordo.

Para utilizar essas classes vamos adicionar uma variável na seção *Var* do formulário que referencie a classe *TReadingThread*, como a seguir:

```
var
  Form1: TForm1;
  rt: TReadingThread = nil;
```

E então no evento *onConnected* do *TIdTCPClient* instanciamos essa *thread*:

```
  rt := TReadingThread.Create(tcClient);
```

E ao encerrar a conexão é preciso também encerrar a *Thread* de leitura, isso é feito no evento *onDisconnected*, como é mostrado a seguir:

```
  if rt <> nil then
  begin
    rt.Terminate;
    rt.WaitFor;
    FreeAndNil(rt);
  end;
```

Observe na **Listagem 8** que na *procedure TLog. DoSynchronize* estamos lidando com o retorno do servidor e se a conexão for bem sucedida chamamos o método *LoginDeuCerto* da classe *TForm1*, que é o formulário. Agora ao vermos o que esse método faz, vemos que o *client* envia outra mensagem para o servidor, solicitando a lista de suporte. A mensagem e enviada usando o método *WriteLn*:

```
  PainelLogin.Visible := false;
  Caption := 'DevChat - '+ EditLogin.Text;
  tcClient.Socket.WriteLn('listaSuporte||'+
IntToStr(IdCliente));
```

Quem efetivamente faz a atualização da lista de membros do suporte técnico é a função *AtualizaListaSuporte* apenas verifica se o código do usuário de suporte passado como parâmetro já existe no *ClientDataset*, se não existir ele é adicionado. Veja na **Listagem 10**.

Montando a aplicação do suporte

As duas aplicações cliente são bastante semelhantes, diferindo apenas em alguns detalhes e funcionamentos. Por isso podemos clonar a aplicação criada no tópico anterior e fazer pequenas modificações. Adicione um novo projeto *Win32* ao grupo e deixe o formulário principal semelhante o formulário do *client* destinado aos clientes, com a exceção que não teremos nele o *TEdit* referente o código da empresa. Crie também as classes *TReadingThread* e *TLog*, como

foi feito no *client* dos clientes.

A primeira coisa que devemos pensar é que precisamos carregar a lista de clientes em nossa aplicação. Ao se conectar é preciso obter a lista dos clientes cadastrados em nosso banco de dados, para isso vamos utilizar o comando *listaCliente* do nosso protocolo de comunicação. O *server* ao receber esse comando executa uma instrução SQL no *BD* que retorna todos os clientes cadastrados no serviço de mensagens, incluindo nas informações o *status* de cada um e devolve isso ao *client* do suporte. Esse por sua vez trata a mensagem no método *DoSynchronize* da classe *TLog*, como é visto na **Listagem 11**.

Veja que é bastante semelhante ao que vimos no aplicativo destinado aos clientes. A diferença principal é que estamos chamando o método *AtualizaListaCliente* que por sua vez se utiliza de um *ClientDataSet* para armazenar os clientes retornados, seguindo o padrão empregado no *client* dos clientes.

> **Nota:** Para que o *client* de suporte se conecte é utilizado o comando *loginSuporte* do protocolo que realiza as conferências necessárias para saber se o usuário de suporte é válido ou não.

Se executarmos a aplicação nesse momento, rodando primeiramente o servidor e posteriormente os dois clientes, veremos que em cada uma teremos a lista de usuários/suporte necessários. Porém como faremos para notificar a mudança de estado, já que mesmo estando conectados os usuários representados nas listas ainda estão *off-line*?

Vamos utilizar o comando *mudancaEstado*. Nele informamos o *Id* do usuário, não importando se é cliente ou suporte, seguidos do seu *status* e se está *on-line*. O *server* por sua vez, ao receber essa mensagem, realiza uma operação de *update* no banco de dados, logicamente para registrar a mudança de estado e então

Listagem 9. Implementando TReadingThread e TLog

```
constructor TReadingThread.Create(AConn: TIdTCPConnection);
begin
  FConn := AConn;
  inherited Create(False);
end;
procedure TReadingThread.Execute;
begin
  while not Terminated and FConn.Connected do
  begin
    TLog.AddMsg(FConn.IOHandler.Readln);
  end;
end;
{ TLog }
class procedure TLog.AddMsg(const AMsg: string);
begin
  with Create(AMsg) do
    try
       Synchronize;
    finally
       Free;
    end;
end;
constructor TLog.Create(const AMsg: string);
begin
  FMsg := AMsg;
  inherited Create;
end;
```

Listagem 10. Código da função AtualizaListaSuporte

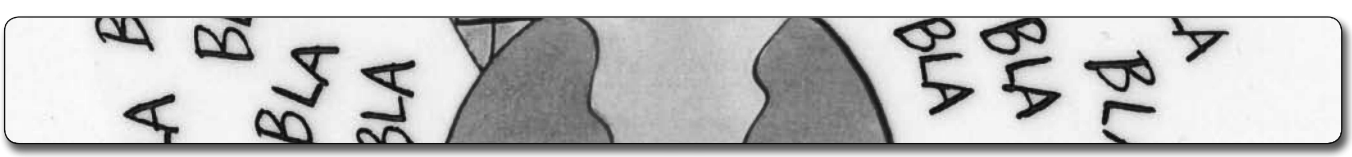
```
procedure TForm1.AtualizaListaSuporte(CodigoSuporte:
  integer; NickSuporte: string; OnLineSuporte,
  StatusSuporte: integer);
begin
  if cdsSuporte.Locate('idSuporte',CodigoSuporte,[]) then
     cdsSuporte.Edit
  else
    cdsSuporte.Insert;
  cdsSuporteidSuporte.AsInteger := CodigoSuporte;
  cdsSuporteNick.AsString := NickSuporte;
  cdsSuporteOnline.AsInteger := OnLineSuporte;
  cdsSuportestatus.AsInteger := StatusSuporte;
  cdsSuporte.Post;
end;
```

Listagem 11. Trecho de DoSynchronize para tratamento da lista de clientes

```
[…]
  if comando = 'listaCliente' then
  begin
    delete(FMsg,1,pos('||',FMsg)+1);
    CodigoCliente := strToInt(Copy(FMsg,1,pos('||',FMsg)-1));
    delete(FMsg,1,pos('||',FMsg)+1);
    NickCliente := Copy(FMsg,1,pos('||',FMsg)-1);
    delete(FMsg,1,pos('||',FMsg)+1);
    OnLineCliente := strToInt(Copy(FMsg,1,pos('||',FMsg)-1));
    delete(FMsg,1,pos('||',FMsg)+1);
    StatusCliente := strToInt(Copy(FMsg,1,pos('||',FMsg)-1));
    DevChatSuporteF.AtualizaListaCliente(CodigoCliente, NickCliente, OnLineCliente,
StatusCliente);
  end;
[...]
```

Listagem 12. Trecho de DoSynchronize com mudança de status

```
[...]
if comando = 'mudancaEstado' then
 begin
    delete(Texto,1,pos('||',Texto)+1);
    CodigoUsuario := strToInt(Copy(Texto,1,pos('||',Texto)-1));
    delete(Texto,1,pos('||',Texto)+1);
    StatusUsuario := strToInt(Copy(Texto,1,pos('||',Texto)-1));
    delete(Texto,1,pos('||',Texto)+1);
    OnLineUsuario := strToInt(Copy(Texto,1,pos('||',Texto)-1));
    NotificarMudancaEstado(AContext,codigoUsuario,TStatus(StatusUsuario),TOnLine(Online
Usuario));
 end;
[…]
```



Listagem 13. Atualizando o status do usuário ao receber comando

```
procedure TDevChatSuporteF.AtualizaEstado(CodigoCliente, Estado,
  Online: integer);
begin
  if CodigoCliente = IdSuporte then exit;
  cdsClientes.First;
  if cdsClientes.Locate('idCliente',CodigoCliente,[]) then
  begin
    cdsClientes.Edit;
    cdsClientesOnline.AsInteger := OnLine;
    cdsClientesStatus.AsInteger := Estado;
    cdsClientes.Post;
  end
  else
  begin
    tcClient.Socket.WriteLn('listaCliente||'+ IntToStr(IdSuporte));
  end;
end;
```

Figura 6. Mudança de status em execução

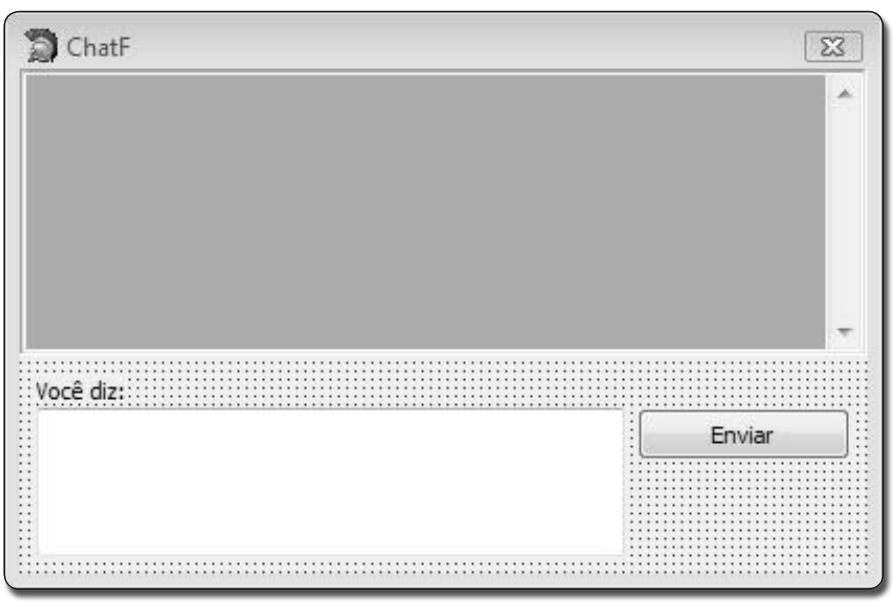


Figura 7. Formulário de conversação

encaminha para todos a mensagem de que o determinado usuário mudou seu *status*. Veja **Listagem 12**.

> **Nota**: Como foi visto no início, os clientes e suporte ficam armazenados em tabelas distintas, então para evitar conflitos de *Id*, ambas as tabelas poderiam utilizar o mesmo *generator* ou a tabela de clientes poderia iniciar a contagem de *Ids* a partir do número 1000.

A procedure *DoSynchronize* consegue saber o estado de um usuário por que fica o tempo todo *escutando* as mensagens que são transmitidas e recebidas entre *Server* e *Client*. Em outras palavras, cada *client* ao receber o comando *mudancaEstado*, atualiza o estado do usuário, como é mostrado na **Listagem 13** e na **Figura 6.**

Desconectando do servidor

No evento *onClose* dos *clients* disparamos o método *tcClient.Disconnect* que por sua vez dispara no server o evento *onDisconect*, indicando que um *client* foi desconectado, dessa forma nesse evento podemos mudar o estado desse cliente para *off-line* e notificamos os outros que ainda estão conectados, como é visto a seguir:

```
Total := Total - 1;
SetStatusUsuario(TUsuario(AContext.Data).
  IdUsuario,TStatus(0),
  TOnLine(0),TUsuario(AContext.Data).Suporte);
if Total > 0 then
 NotificarMudancaEstado(AContext,TUsuario(
  AContext.Data).IdUsuario,
    TStatus(0),TOnLine(0));
```

Nota do DevMan

O que são *Generators's*?

O *Generator* um recurso existente no banco de dados *Firebird* capaz de gerar número seqüenciais. Quem chegou a utilizar os bancos de dados *Paradox* e *DBase* conhece esse recurso como auto-numeração. Havia um tipo de campo nesses bancos de dados que fazia a geração automática de números seqüenciais que normalmente são usado para evitar duplicação de registros da base.

Com a entrada do *Firebird e Interbase* ao mercado, esse conceito morreu dando lugar aos *Generator's*. No *Firebird 2.0* o *Generator* foi substituído pelo recurso *Sequence*, que tem a mesma finalidade.

Trocando mensagens

Após termos o controle de quem está ou não está conectado é preciso realizar a troca de mensagens entre *clients*. Antes de tudo é preciso entender como o *chat* se dará. O cliente dará um clique duplo sobre o usuário de suporte que está *on-line* e então a janela de *chat* aparece. Caso o cliente dê um clique duplo em um usuário de suporte com uma conversa já iniciada, a janela onde essa conversa está sendo mantida é então exibida e não será criada uma nova. Tendo essas premissas, vemos que é preciso então criar uma forma de controlar quais janelas que já estão abertas.

Para tal começaremos com o *client* dos clientes. Adicione a esse projeto um novo formulário, salve-o como *ChatU*.pas e deixe-o como na **Figura 7**. E no evento *onClick* do botão *Enviar* disparamos o comando de envio de mensagens (**Listagem 14**).

Perceba que é simples. Apenas enviamos a mensagem para o componente *TCPClient* usando o método *WriteLn*. Em seguida atualizamos o componente *Memo* com informações de data e hora da mensagem.

Crie uma propriedade pública, de nome *NomeSuporte* e no seu método *Set* mude o *Caption* da janela como segue:

```
FNomeSuporte := Value;
Caption := 'Conversando com : ' + Value;
```

Vamos adicionar um campo privado no formulário principal do *client*, um campo de nome *ListaJanelas* que será do tipo *TList*. No evento *onCreate* do formulário instanciamos esse campo. Essa lista irá armazenar as conversas que estão em andamento.

```
ListaJanelas := TList.Create;
```

Para verificar se já existe uma conversa em andamento vamos realizar um truque aqui. Utilizaremos a propriedade *Tag* do *TForm* para armazenar o *id* do suporte com quem se está conversando e vamos procurar por esse *id* na lista de janelas. Se o *id* não constar criamos um novo formulário, caso contrário retornamos a instância encontrada. Tudo isso pode ser visto na **Listagem 15**. Esse método é então utilizado no evento *OnDblClick* do *DBGrid* de usuários de suporte disponíveis.

Listagem 14. Código do evento OnClick do botão Enviar

```
procedure TChatF.btEnviarClick(Sender: TObject);
var
 I: integer;
begin
 for I := 0 to memoTexto.Lines.Count - 1 do
  Form1.tcClient.Socket.WriteLn('mensagemParaSuporte||'+
  IntToStr(Form1.IdCliente)+'||'+IntToStr(Self.Tag)+
  '||'+(memoTexto.Lines[I])+'||'+IntToStr(I)+'||');
 memoChat.Lines.Append(DateTimeToStr(now) + ' - Você diz:' +
  MemoTexto.Text);
 memoTexto.Clear;
 memoTexto.SetFocus;
end;
```

Listagem 15. Controlando quais conversas já estão iniciadas

```
function TForm1.CriaFormChat(ParaIdSuporte: integer): TChatF;
var
 x: integer;
 encontrada: boolean;
 Form : TChatF;
begin
  encontrada := false;
  for x := 0 to ListaJanelas.Count - 1 do
  begin
    if TForm(ListaJanelas.Items[x]).Tag = ParaIdSuporte then
    begin
      encontrada := true;
      result := TChatF(ListaJanelas.Items[x]);
      break;
    end;
  end;
  if not encontrada then
  begin
    Form := TChatF.Create(Self);
    Form.Tag := ParaIdSuporte;
    ListaJanelas.Add(Form);
    result := Form;
  end;
end;
```

O método *IniciaChatCom* localiza o registro do usuário de suporte com quem estamos tentando falar e verifica se ele já possui uma conversa em aberto conosco, se possuir a janela de conversação é exibida. Quando o parâmetro *Mensagem* possui algum valor ele é então exibido na janela de conversação. Vale lembrar aqui que a mensagem que será exibida é enviada primeiramente ao *Server* e que ele é que encaminha para o destinatário.

O comando é desmembrado para suas respectivas partes e então o método *EnviaMensagemParaSuporte* é utilizado. Esse método tem a tarefa de percorrer a lista de *clients* conectados e identificar quem deve receber a mensagem e então passá-la. Na **Listagem 16** vemos isso. Essa lista de *clients* é retornada pelo método *LockList* de *TIdContext*.

Recebendo uma mensagem

Podemos dizer agora que nossa solução está quase pronta. Precisamos ver o último passo, receber e exibir as mensagens. Ao receber um comando *MensagemParaCliente* é preciso incluir no método *TLog.DoSynchronize* o respectivo tratamento. Observe na **Listagem 17** que utilizamos o método *IniciaChatCom* passando todos os parâmetros, o que fará com que a mensagem recebida seja exibida corretamente. E novamente não há grande complexidade no código. Novamente desmembramos a mensagem recebida e extraímos dela o comando recebido. Por fim chamamos a função *IniciaChatCom*.

Avisando que você está digitando

Um recurso interessante é avisar para ambas as parte, de que a pessoa na outra ponta está digitamdo algo, assim como o *MSN*. Para implementar isso, vamos fazer uso do comando *Escrevendo* ou

Listagem 17. Recebendo uma mensagem

```
  if comando = 'mensagemParaCliente' then
  begin
    delete(FMsg,1,pos('||',FMsg)+1);
    CodigoSuporte := strToInt(Copy(FMsg,1,pos('||',FMsg)-1));
    delete(FMsg,1,pos('||',FMsg)+1) ;
    delete(FMsg,1,pos('||',FMsg)+1) ;
    Mensagem := Copy(FMsg,1,pos('||',FMsg)-1);
    delete(FMsg,1,pos('||',FMsg)+1);
    NumeroLinha := strToInt(Copy(FMsg,1,pos('||',FMsg)-1));
    Form1.IniciaChatCom(CodigoSuporte,Mensagem, NumeroLinha);
  end;
```

Listagem 18. Avisando da digitação

```
procedure TChatF.memoTextoChange(Sender: TObject);
begin
  if Trim(memoTexto.Text) = '' then
  begin
    FDigitando := false;
    Form1.tcClient.Socket.WriteLn('pareiEscrever||'+
      IntToStr(Form1.IdCliente)+'||'+IntToStr(Self.Tag)+'||');
    exit;
  end;
  if not FDigitando then
  begin
    FDigitando := true;
    Form1.tcClient.Socket.WriteLn('Escrevendo||'+
      IntToStr(Form1.IdCliente)+'||'+IntToStr(Self.Tag)+'||');
    exit;
  end;
end;
```

Listagem 19. Tratando comando Escrevendo/pareiEscrever

```
procedure TLog.DoSynchronize;
var
{...}
begin
{...}
  if comando = 'Escrevendo' then
  begin
    delete(FMsg,1,pos('||',FMsg)+1);
    CodigoSuporte := strToInt(Copy(FMsg,1,pos('||',FMsg)-1));
    Form1.SuporteEscrevendo(CodigoSuporte,true);
  end;
  if comando = 'pareiEscrever' then
  begin
    delete(FMsg,1,pos('||',FMsg)+1);
    CodigoSuporte := strToInt(Copy(FMsg,1,pos('||',FMsg)-1));
    Form1.SuporteEscrevendo(CodigoSuporte,false);
  end;
end;
procedure TForm1.SuporteEscrevendo(idSuporte: integer; Escrevendo: boolean);
begin
  cdsSuporte.Locate('IdSuporte',idSuporte,[]);
  with CriaFormChat(idSuporte) do
  begin
    if Escrevendo then
      BarraMensagem.SimpleText := cdsSuporteNick.AsString + ' está escrevendo...'
    else
      BarraMensagem.SimpleText := '';
  end;
end;
```

Listagem 16. Enviando a mensagem para o Client correto

```
procedure TServerF.EnviaMensagemParaSuporte(const AContext:
  TIdContext; idUsuarioOrigem, idSuporte: integer; Mensagem: string; Linha: integer);
var
 List: TList;
 I: integer;
 HackContext : TContextHack;
begin
  List := TContextHack(AContext).FContextList.LockList;
  try
    for I := 0 to List.Count - 1 do
    begin
      HackContext := TContextHack(List[i]);
      if TUsuario(HackContext.Data).IdUsuario = idSuporte then
      begin
         HackContext.Connection.IOHandler.WriteLn('mensagemParaSuporte||'+
           IntToStr(idUsuarioOrigem)+'||'+IntToStr(idSuporte)+
           '||'+Mensagem+'||'+IntToStr(Linha)+'||');
         break;
      end;
    end;
  finally
    TContextHack(AContext).FContextList.UnlockList;
  end;
end;
```

PareiEscrever definidos anteriormente no protocolo de comunicação.

Vamos adicionar no formulário de *chat* um *TStatusBar* e no evento *OnChange* do *TMemo* de digitação nós verificamos se existe texto ainda no *TMemo*, se existir verificamos se já não enviamos o comando *Escrevendo*. Se o comando ainda não foi enviado então o enviamos. Caso o *TMemo* não tenha texto algum, enviamos o comando *PareiEscrever*, confira tudo isso na **Listagem 18**. Quando o *client* recebe esses comandos ele apenas limpa ou adiciona o texto informando isso no *TStatusBar* (**Listagem 19**).

Por fim, será necessário fazer a implementação do que acabamos de ver, na versão *client* de uso do suporte técnico. Ele é muito similiar ao *Client* destinado para o clientes. Existem algumas diferenças como, por exemplo, a de ao invés de listar os usuários de suporte, os clientes é que são listados.

Na seção de download dessa edição, será possível e analisar o código-fonte completo dessas aplicações.

Conclusão

Este artigo foi apenas uma introdução no desenvolvimento de um *Messenger*. A partir do que foi mostrado, novos recursos podem ser explorados como envio de arquivos, avisos de chamada de atenção. No *server* um *log* e histórico poderia ser montado além de ter a possibilidade de convertê-lo para um serviço do Windows. Fazer tudo isso exigiria várias edições, porém estou à disposição quando alguma dúvida surgir. Muito obrigado. ●

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

A infra-estrutura de TI que você precisa.

A Alog trabalha 24 horas por dia, 7 dias por semana, 365 dias por ano para garantir aos seus clientes corporativos estabilidade, agilidade, redundância, escalabilidade, e mais do que tudo, **tranqüilidade**. Entre em contato e saiba por que a Alog é líder em hosting gerenciado

ALOG
DATA CENTERS DO BRASIL

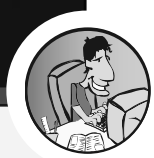
HOSTING GERENCIADO | COLOCATION | CONTIGÊNCIA
E-MAIL CORPORATIVO | CONECTIVIDADE | SERVIÇOS PROFISSIONAIS

0800 282 3330 | alog.com.br

Nesta seção você encontra artigos sobre técnicas que poderão aumentar a qualidade do desenvolvimento de software

Aplicando MVC em Delphi

Aprenda a aplicar este poderoso pattern em suas aplicações



Resumo DevMan

O padrão de projeto MVC surgiu pela primeira vez em 1979 no projeto Smalltalk na Xerox por Trygve Reenskaug, e na minha opinião foi uma das maiores contribuições na área do desenvolvimento de softwares de todos os tempos pelo menos para a época em que surgiu. De lá pra cá diversas variações do MVC surgiram e a mais conhecida é a MVP (Model View Presenter) que devido a falta de material consistente sobre como o MVC tem que ser, causa grande confusão até em programadores experientes.

Nesse artigo veremos

- Padrões de Projeto;
- MVC;
- Boas Práticas;
- UML;

Qual a finalidade

- Modelar um cadastro de contato utilizando o padrão estrutural MVC como realmente ele deve ser, aplicando padrão de projeto Observer.

Quais situações utilizam esses recursos?

- Mais do que separar em camadas, o MVC deve ser utilizado quando se pretender aumentar o nível de abstração de uma aplicação, diminuir o forte acoplamento entre os componentes da mesma, porém mantendo a comunicação entre eles.

Rodrigo Carreiro Mourão
(rodrigocarreiro@rmfactory.com.br)
apaixonado por tecnologia da informação. Atualmente é sócio diretor da RM Factory Soluções em TI, empresa especializada em consultoria na área de TI e desenvolvimento de softwares customizados, parceira da TDS Tecnologia - RJ.Possui os títulos de BDS2006 Win32 Product Certified e Borland Instructor Certified. Como instrutor de treinamentos oficiais já ministrou treinamentos para centenas de profissinais e servidores de orgãos como TJ-RJ, JF, BB, IBGE, MB, EB, etc. Palestrante da Borland Conference 2007. Matenedor da comunidade www.delphisophp.com, a primeira do Brasil 100% feita com Delphi for PHP. Autor de diversos artigos técnicos veiculados nos principais sites e revistas sobre Delphi do Brasil.

Nos últimos anos, temos visto um aquecimento em todas as áreas da economia e a área de TI tem um papel importantíssimo nesse crescimento, pois independente do setor todos estarão direta ou indiretamente ligados a ela. Com todo esse crescimento as necessidades também aumentam a demanda por softwares de qualidade da mesma forma e a única coisa que diminui é o prazo que já era curto e agora está ainda mais apertado.

Não há para onde escapar. As empresas de desenvolvimento de softwares têm contratado consultorias especializadas para poder atender a demanda de seus clientes. Seus softwares, que outrora atendiam, já não atendem mais. Com isso (isso digo por experiência própria), são apresentados a essas empresas e suas equipes técnicas, padrões e boas práticas de desenvolvimento que eram desconhecidas por eles. Digo isso para poder *justificar* o porquê de se ouvir

falar tanto em padrões de projeto, boas práticas, OO, etc. Como disse *Steve Jobs* certa vez:

Você não pode simplesmente perguntar ao usuário o que ele quer e tentar dar-lhe isso, quando você conseguir terminar o produto o usuário estará querendo outra coisa.

Eu vejo essa mudança de cenário com bons olhos, aliás com ótimos olhos, pois isso elevará o nível das empresas brasileiras, de sua equipe e conseqüentemente o nível da *população* de TI do Brasil.

Não podemos ficar fora deste cenário, por isso veremos neste artigo como introduzir nossas aplicações feitas em Delphi nesse novo cenário. Como aplicar em nossas aplicações essas técnicas e boas práticas e assim aumentar o nível de nossas aplicações? Óbvio que falar em boas práticas é falar de diversas soluções reutilizáveis para construção de softwares orientados a objetos eficientes e que em uma única edição não é possível abordar, assim o foco deste artigo será o *pattern arquitetural MVC* em conjunto com o padrão *Observer*. Assim vamos ao que interessa.

Padrões de Projeto

O conceito de padrão de projeto foi usado pela primeira vez na década de 70 pelo arquiteto e urbanista austríaco *Christopher Alexander*. Ele observou na época que as construções, embora fossem diferentes em vários aspectos, todas passavam pelos mesmos problemas na hora de se construir. Eram problemas que se repetiam em todas elas e na maioria das vezes numa mesma fase da construção.

Foi ai que *Christopher* resolveu documentar esses problemas e mais do que isso, passou também a documentar as soluções que eram aplicadas para resolução destes problemas. Entenda que até agora não há nada de programação ou informática e sim projetos, plantas, métricas, definições. Nesse momento surgiam os padrões de projetos para a engenharia civil, padrões esses que descreviam os problemas recorrentes em um projeto de engenharia e a solução reutilizável para este problema. Em seus livros *Notes on the Synthesis of Form, The Timeless Way of Building e A Pattern Language Christopher* estabelece que um padrão deva ter as seguintes características:

- Encapsulamento;
- Generalidade;
- Equilíbrio;
- Abstração;
- Abertura;
- Combinatoriedade.

Ou seja, um padrão deve ser independente. Deve permitir a construção de outras realizações. Deve representar abstrações do conhecimento cotidiano. Deve permitir a sua extensão para níveis mais baixos de detalhe e deve ser relacionado hierarquicamente. E assim, foram definidos os padrões para projetos na engenharia civil.

Anos mais tarde *Erich Gamma, Richard Helm, Ralph Johnson* e *John Vlissides* iniciaram suas pesquisas baseados nos trabalhos de *Alexander*. Também conhecidos como *GOF* (*Gang of Four*) eles começaram a descrever e documentar os problemas e soluções para desenvolvimento de softwares orientados a objetos e em 1995 lançaram o livro que ser tornou um fenômeno na área de análise e desenvolvimento de software: *Design Patterns: Elements of Reusable Object-Oriented Software*.

Neste momento iniciava-se uma nova fase no desenvolvimento de sistemas, pois agora havia um padrão a ser seguido, e cada padrão apresentando uma solução que poderia se reutilizada várias vezes para solucionar aqueles problemas recorrentes no desenvolvimento de software. Os padrões *GOF*, como são conhecidos, se dividem em três grandes categorias: *criacionais, estruturais* e *comportamentais* somando 23 no total.

Obviamente que o intuito deste projeto não é focar em Padrões de Projeto, isso poderá ser feito em artigos posteriores onde poderemos demonstrar como aplicá-los em Delphi, porém gostaria de rapidamente falar sobre interfaces que são os pilares dos padrões de projetos e que por sinal faremos dela, ao programarmos, nosso exemplo MVC.

Nota do DevMan

Os 23 padrões do GOF são:

Padrões Criacionais
- *Abstract Factory*
- *Builder*
- *Factory Method*
- *Prototype*
- *Singleton*

Padrões Estruturais
- *Adapter*
- *Bridge*
- *Composite*
- *Decorator*
- *Façade*
- *Flyweight*
- *Proxy*

Padrões Comportamentais
- *Chain of Responsability*
- *Command*
- *Interpreter*
- *Iterator*
- *Mediator*
- *Memento*
- *Observer*
- *State*
- *Strategy*
- *Template Method*
- *Visitor.*

Em OO temos a herança como recurso para reaproveitar códigos e facilitar a manutenção, mas a herança em algumas linguagens como o PHP restringem-se a uma única classe. Não podemos herdar de duas ou mais classes ao mesmo tempo. Há linguagens como o C que permitem o uso de herança múltipla, porém os desenvolvedores do Delphi (pascal) optaram pelo uso de interfaces.

Interfaces nada mais são do que uma espécie de regra que a classe que a implementa deverá seguir ou ter em sua estrutura. Voltando àquela analogia básica que sempre usamos em OO, uma classe é como a planta de uma casa, algo ainda intangível, no papel e o objeto é a casa construída, algo concreto. Assim podemos aqui colocar a interface como um contrato de licitação onde estarão todas as cláusulas que ditam as regras e normas para a construção desse objeto.

Não importa qual construtora irá

construir a casa, desde que esta construtora se enquadre nas regras do contrato. Assim podemos crer que a casa será construída como foi predeterminada. Na prática podemos criar uma interface em nosso modelo que terá os métodos que julgamos necessários para determinada ação e desta maneira toda a classe que implementar esta interface terá estes métodos e os implementará de acordo com a sua necessidade.

A vantagem é que uma classe pode implementar mais de uma interface e com isso manter um relacionamento *é um* com mais de uma estrutura

MVC – Model View Controller

Separar as aplicações em camadas não é algo novo no meio de desenvolvimento de softwares. A busca por aplicações de fácil manutenção inspirou grandes mentes a desenvolver técnicas e modelos que auxiliam nesta tarefa. Porém de início quero deixar claro um fato importantíssimo: separar uma aplicação em camadas pura e simplesmente não significa que você esta aplicando MVC.

A arquitetura em camadas é utilizada para separar responsabilidades em uma aplicação. Este tipo de aplicação se popularizou muito no início da década de 90 com o *boom* da internet, porém muitos desenvolvedores ainda não conhecem a técnica a fundo, devido a escassez de documentação eficiente sobre este padrão arquitetural. Junto com a disseminação da arquitetura em camadas, veio o ressurgimento do modelo MVC criado em *Smalltalk* e que traz simplicidade e coerência à *View*.

Tanto o modelo em camadas quanto o MVC são padrões arquiteturais muito similares e que passaram a ser continuamente confundidos. MVC e desenvolvimento em camadas são abordagens diferentes que podem ser ou não aplicados em conjunto.

A abordagem do desenvolvimento em camadas visa organizar e manter os componentes separados baseados em algum critério que na maioria das vezes é a função que este componente desempenha. Separando os componentes em grupo, conseguimos diminuir o acoplamento entre eles isolando assim a mudanças que são realizadas em uma camada para que não afetem as outras. Uma aplicação Win32 desenvolvida 100% orientada a objetos poderia sem problema algum ser separada em camadas para facilitar o desenvolvimento e manutenção. Observe o diagrama da **Figura 1** e tente identificar as camadas.

Nota: O emprego correto do termo componente é utilizado para descrever um artefato de software que pode ser uma classe, objeto, camada, etc. Nós, que estamos familiarizados com a VCL do Delphi, tendemos a associar o termo componente com os controles utilizados na VCL. Estes últimos são identificados pelo termo *controle*. Assim, estarei aqui me referindo a artefatos de software como componente.

A separação em camadas não se dá pela separação *física* das classes, mas sim pela separação *lógica* de acordo com um critério definido pelo *analista/arquiteto* de sistemas. No caso exposto na **Figura 1** as camadas (lógicas) estão divididas segundo suas responsabilidades, na **Figura 2** isso fica mais claro. Observe:

Bem se separar em camadas não é MVC então o que é? Digamos que temos uma aplicação onde aplicamos a separação em camadas de acordo com a responsabilidade de cada componente. Se estes componentes estão separados, ou melhor, desacoplados então temos que de alguma forma fazer com que se comuniquem, mas que continuem independentes em sua essência. Isso realmente seria excelente, é o famoso *cada um na sua, mas com alguma coisa em comum*. Este tipo de abordagem torna o modelo muito eficiente. Mas como manter as interfaces gráficas atualizadas refletindo o atual estado do modelo com quem ela interage?

Deu para perceber com o questionamento que entre outras coisas o modelo MVC tem a ver com sincronização entre os componentes? As interfaces gráficas geralmente exibem o estado dos objetos de uma aplicação, e isso deve ser feito em tempo real. Qualquer alteração no objeto

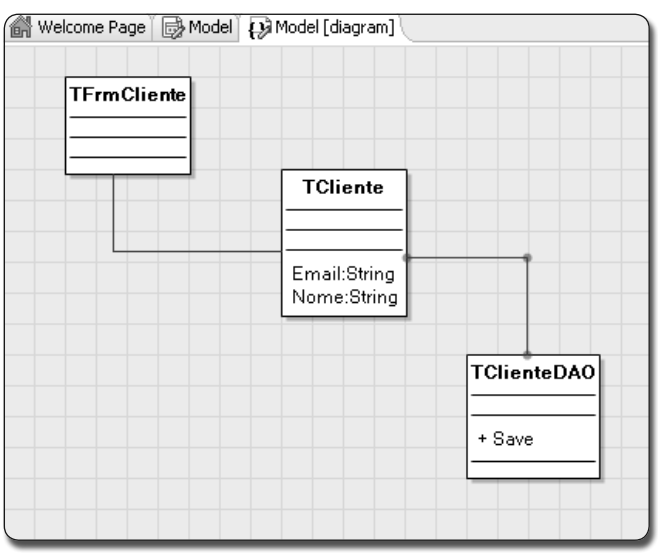


Figura 1. Modelo UML

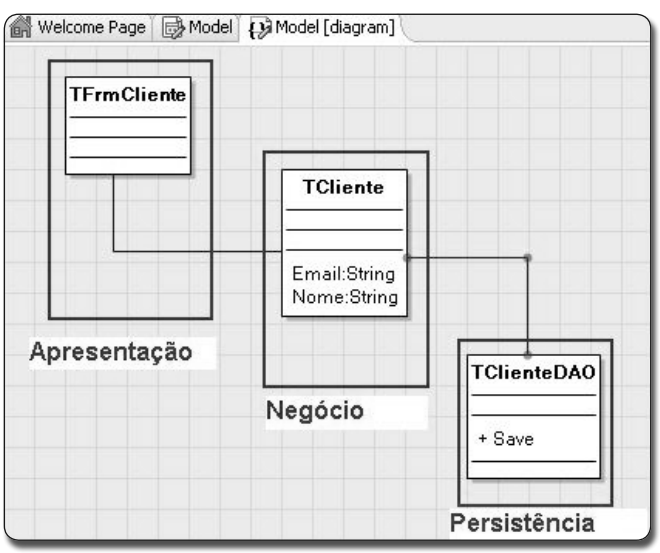


Figura 2. Modelo Separado em Camadas

deverá ser refletida na hora, na *View*.

Outro ponto é que o usuário da sua aplicação deverá interagir com o seu modelo de negócio e ele o vai fazer através da *View*. Daí, vemos importância do MVC aplicado a softwares desenvolvidos em Delphi, pois por ser RAD o Delphi nos trás uma produtividade muito grande, porém isso tem um lado ruim. Se cada controle em um *form* for responsável por invocar métodos nos objetos o código tende a ficar repetitivo e difícil de manter. Diversas regras e rotinas poderão estar contidas dentro dos botões e demais controles no formulário. Isso torna o código poluído, de difícil manutenção e altamente acoplado, pois suas regras de negócio estão dispersas entre o modelo e a *View*.

É neste cenário que entra o MVC, para além de termos as camadas com suas responsabilidades, temos os componentes da aplicação interagindo entre si em tempo real onde o modelo passa a ser ativo notificando as *views* inscritas nele. Este modelo consiste no bom uso integrado de alguns *Design Patterns* (Padrões de Projeto) clássicos, como *Observer* e *Strategy*.

Se buscarmos na internet ou em livros o significado de MVC, vamos encontrar algo como:

Model-view-controller (MVC) é um padrão de arquitetura de software. Com o aumento da complexidade das aplicações desenvolvidas torna-se fundamental a separação entre os dados (Model) e o layout (View). Desta forma, alterações feitas no layout não afetam a manipulação de dados, e estes poderão ser reorganizados sem alterar o layout.

O model-view-controller resolve este problema através da separação das tarefas de acesso aos dados e lógica de negócio, lógica de apresentação e de interação com o utilizador, introduzindo um componente entre os dois: o Controller. MVC é usado em padrões de projeto de software, mas MVC abrange mais da arquitetura de uma aplicação do que é típico para um padrão de projeto.

No modelo em camadas é comum dividir a aplicação em: apresentação (interface), domínio (*Model*) e acesso a dados. Podemos dizer que a, grosso modo, em MVC a camada de apresentação também é separada em *View* e *Controller*.

Geralmente o fluxo de iteração entre os componentes no MVC se dá na maioria das vezes da seguinte maneira: O usuário interage com a interface de alguma forma, clicando em um botão da *View* (no nosso caso *Form*). O *Controller* é quem recebe o estimulo provocado na *View*, acessando o *Model*, e invocando um método ou atualizando de alguma forma. O *Model* por sua vez notifica a(s) *View's* inscritas neles para receber as atualizações. A *View* recebe a notificação e se encarrega de atualizar seus controles conforme a necessidade. Os diagramas das **Figuras 3** e **4** expressam a seqüência dos acontecimentos.

O estímulo vindo do usuário (ou de outro componente se você está usando MVC fora de uma interface gráfica) vai para o *Controller* que é o componente com inteligência o suficiente para saber qual operação invocar no *Model*. A operação invocada pode efetuar a mudança de estado no *Model*. Como a *View* observa este, assim que a mudança de estado for realizada ela é atualizada.

Assim vemos que o diferencial do MVC para o modelo separado em camadas é que o primeiro foca em mais do que separar em camadas, mais cuida da iteração entre os componentes que fazem parte do modelo.

MVC no Delphi

Vamos então aplicar o conceito MVC em um pequeno exemplo feito em Delphi. Nosso exemplo focará em criar o modelo MVC. No próximo artigo daremos seqüência ao exemplo adicionando a camada de persistência e concluído assim nosso exemplo.

Nota do DevMan

Em MVC geralmente as camadas são descritas assim:

Model: classes de domínio no seu modelo de negócio. Representa o estado do sistema.

View: parte exposta, renderiza o Model em uma forma especifica para interação (WebForm, Form, HTML, etc.)

Controller: Processa e responde a eventos, geralmente ações do usuário, invocando alterações no Model.

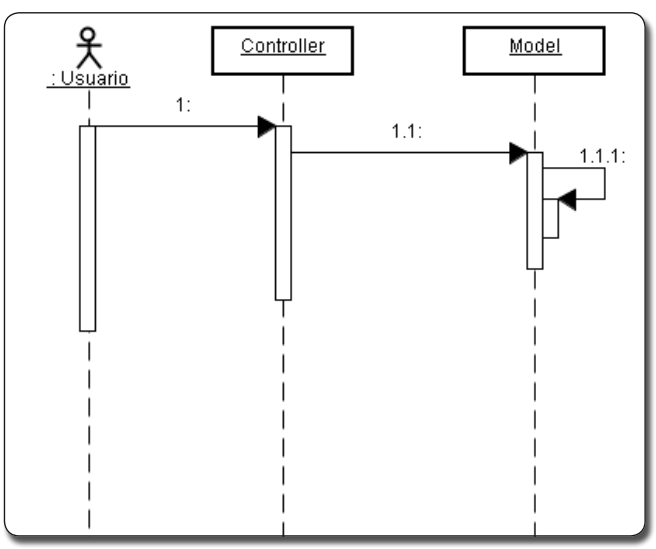


Figura 3. Usuário interage com a view e o controller repassa ao modelo

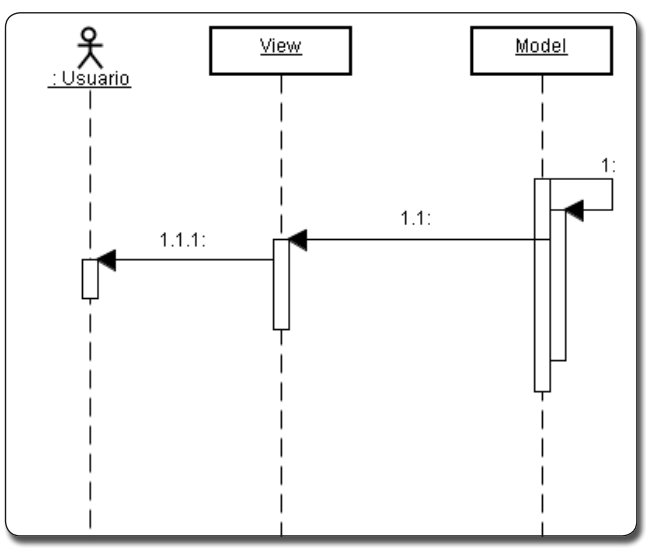


Figura 4. O modelo notifica e View que é atualizada

Como visto anteriormente, um dos desafios da arquitetura em camadas é justamente manter a camada de apresentação sincronizada com a camada de negócio. No MVC isso não é um desafio e sim parte do modelo e isso é feito aplicando o padrão de projeto *Observer*. O padrão *Observer* define uma dependência um-para-muitos entre objetos de forma que se um mudar seu estado, todos os objetos dependentes serão notificados e atualizados automaticamente. [GoF]. Este padrão é muito freqüente em projetos orientados a objetos. No Delphi este padrão está presente na notificação entre os componentes no *form designer*, quando um componente é removido ou inserido.

Em nosso exemplo em Delphi faremos uso do padrão *Observer*, então vamos a ele.

> **Nota**: O exemplo será desenvolvido utilizando a versão 2006 do Delphi, porém com exceção da modelagem UML feita no *Together* o exemplo poderá ser construído com qualquer versão do Delphi a partir da versão 3.

No Delphi 2006 crie uma nova aplicação Win32 e salve-a em uma pasta padrão. Renomeie o formulário para *FrmMenuPrincipal.dpr*. Ainda neste projeto adicione uma nova *Unit* onde iremos criar nosso modelo padrão, ou seja, nossa classe básica assim como as interfaces necessárias ao padrão *Observer*. Salve-a como *BaseModel.pas*. A **Figura 6** demonstra nosso modelo de classe.

As interfaces *IObserver* e *IObservable* são requeridas para o padrão *Observer*. *TBaseObject* será o objeto base para todas as outras classes presente em nossa aplicação. A interface *IController* é apenas uma sugestão que eu costumo utilizar para criar um repositório de *Controller*.

É evidente que utilizando o *Together* após fazermos a modelagem UML não há a necessidade de gerarmos o código, pois o mesmo é gerado automaticamente, porém é possível codificar manualmente para aqueles que possuem outras versões do Delphi. Veja a **Listagem 1** que contém todo o código do diagrama gerado na **Figura 6**. Assim na *Unit BaseModel.pas* digite o código da **Listagem 1**.

No código da **Listagem 1** temos a declaração das interfaces requerida pelo padrão *Observer*. A interface *IObserver* possui apenas um método (*Update*), que deverá ser implementado por toda a classe que quiser ser um observador em nosso modelo. Isso por que o *observado* quando for notificar os observadores irá invocar o método *Update* daí o porquê deste método estar presente ai. O parâmetro do tipo *IObservable* serve para que o observador possa saber por quem ele foi notificado. Observe que na linha um temos apenas um *hint* da interface *IObservable*. Isso é necessário, pois o compilador Delphi é *Top-Down* e a interface *IObservable* é declarada abaixo da interface *IObserver* e como o método *Update* possui um parâmetro do tipo *IObservable* temos que lançar mão da técnica de *forward*. O mesmo ocorre na declaração de *TBaseObject*.

Para incluí-las basta pressionar *Ctrl + Shift + G* dentro da declaração de sua interface. A interface *IObservable* possui 3 métodos, isso porque qualquer objeto que possa ser observado em um modelo deverá ter um canal para que os observadores possam se inscrever para serem notificados (*Atach*), da mesma forma os observadores devem possuir uma maneira de se retirar da lista de notificações (*Desatach*) e por fim um método para notificar os observadores que algo aconteceu. A interface *IController* como citei anteriormente é apenas uma sugestão para que possamos criar uma coleção de *Controllers* para futura manipulação.

Bem interfaces são apenas contratos, regras, declarações que precisarão ser implementados por uma classe concreta. Como no MVC nossos *Model's* são *Observadores* então nossa *BaseObject* é quem vai implementar a interface *IObservable* e nossas *views* a interface *IObserver*. Assim vamos ao código da **Listagem 2** que descreve a nossa *BaseObject* ainda na *Unit BaseModel*.

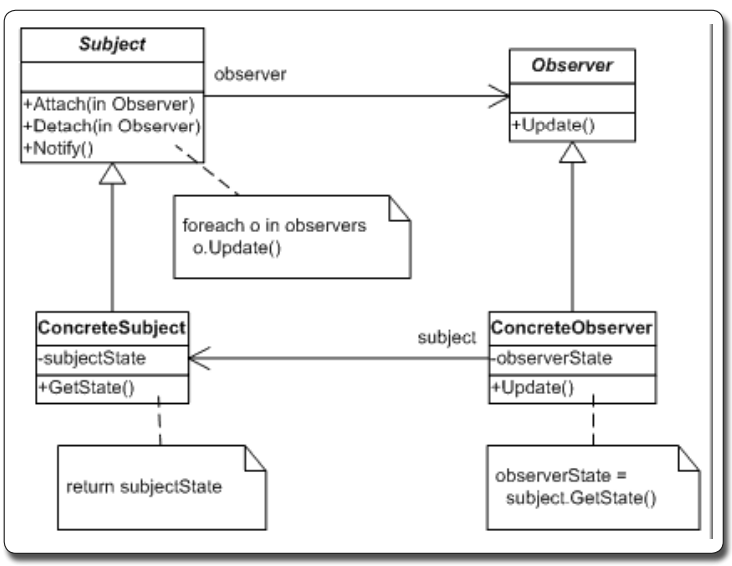


Figura 5. Modelo do padrão Observer

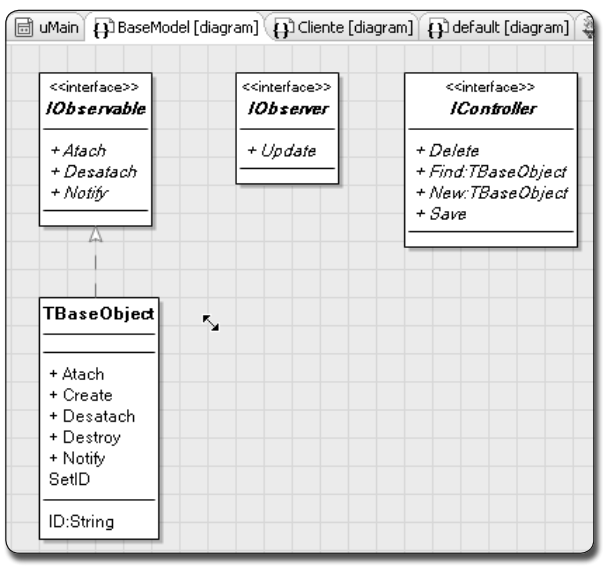


Figura 6. Modelo de classe

Na **Listagem 2** temos a declaração da classe que servirá de base para todas as demais em nosso modelo. A primeira consideração a fazer é em relação ao fato de herdarmos de *TInterfacedObject*. Isso é uma condição para todas as classes que irão implementar uma ou mais interfaces no Delphi, isso se deve ao fato desta classe já implementar 3 métodos essenciais para se trabalhar com interfaces. São eles: *QueryInterface*, *_AddRef* e *_Release*. Existe ainda outras classes que implementam estes três métodos e, portanto poderiam ser usadas aqui também, como *TContainedObject*, *TInterfaceList* e *TInterfacedPersistent*, porém a classe *TInterfacedObject* é a mais básica de todas elas.

Seguindo adiante temos na linha 3 um atributo privado *FObservers* que servirá como contêiner para todos os observadores que se inscreverem em um de nossos modelos. O campo *FID* e o método *setId* são resultado da implementação da propriedade ID. Na seção pública da nossa classe temos além do construtor e destrutor a declaração, para posterior implementação, dos métodos presentes na interface *IObservable* (*Atach*, *Desatach* e *Notify*). Vale ressaltar aqui que somente poderá ser passado como parâmetros para os dois primeiros métodos objetos produtos de uma classe que implemente a interface *IObserver*, ou seja, estamos garantindo que somente observadores se inscrevam na lista para serem notificados e se são observadores logo possuem sem dúvida nenhuma o método *update* que será invocado quando a notificação for disparada. Na **Listagem 3** podemos conferir a implementação dos 5 métodos presentes em nossa *TBaseObject*.

Na **Listagem 3** temos a implementação dos métodos outrora definidos na interface *IObservable* além do destrutor e construtor. O método *Attach* recebe como parâmetro um objeto do tipo *IObserver*, ou seja, todo objeto que implemente a interface *IObserver* poderá ser passado como parâmetro e isso é excelente, pois se uma classe que hoje não é um observador amanhã poderá ser, bastando para isso implementar a interface *IObserver*. E como uma classe pode implementar uma ou mais interfaces não corremos o

Listagem 1. Declaração das Interfaces

```
IObservable = interface;

IObserver = Interface
  ['{65032CAD-7441-4754-A860-BFECE58D50EF}']
  procedure Update(Observable: IObservable);
end;

 IObservable = Interface
   ['{637D22E1-45BD-479E-96D3-39F6DD94234B}']
   procedure Atach(Observer: IObserver);
   procedure Desatach(Observer: IObserver);
   procedure Notify;
end;

TBaseObject = class;

IController = Interface
  ['{2C5753F9-473F-4D22-9FEA-28E2B84C629E}']
  procedure Save;
  function Find(ID: String): TBaseObject;
  function New: TBaseObject;
end;
```

Listagem 2. Declaração da Classe BaseObject

```
TBaseObject = class(TInterfacedObject, IObservable)
private
  FObservers: TInterfaceList;
  FID: String;
  procedure SetID(const Value: String);
protected

public
  constructor Create;
  destructor Destroy; override;
  procedure Atach(Observer: IObserver);
  procedure Desatach(Observer: IObserver);
  procedure Notify;
published
  property ID: String read FID write SetID;
end;
```

Listagem 3. Implementação dos Métodos

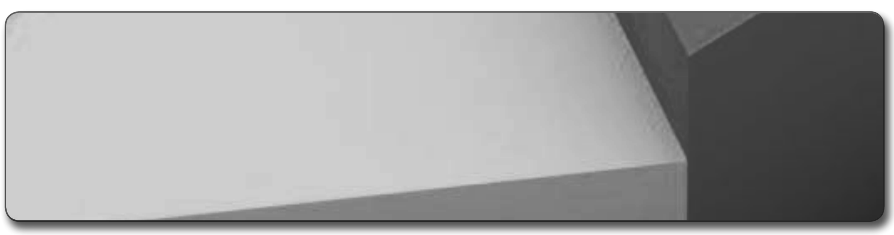
```
procedure TBaseObject.Atach(Observer: IObserver);
begin
  FObservers.Add(Observer);
end;

constructor TBaseObject.Create;
var
  GID: TGUID;
begin
  FObservers := TInterfaceList.Create;
  if CoCreateGuid(GID) = S_OK then
    FID := GUIDToString(GID);
end;

procedure TBaseObject.Desatach(Observer: IObserver);
begin
  FObservers.Remove(Observer);
end;

destructor TBaseObject.Destroy;
begin
  FreeAndNil(FObservers);
  inherited;
end;

procedure TBaseObject.Notify;
var
  Obs: IInterface;
begin
  for Obs in FObservers do
    IObserver(Obs).Update(Self);
end;
```



Listagem 4. Declaração da Classe Cliente

```
uses BaseModel;

type
  TCliente = class(TBaseObject)
  private

  protected

  public
    procedure Salvar;
    function Buscar(Codigo: String): TCliente;
  published
    property Nome: String;
    property Email: String;
    property Telefone: String;
  end;
```

Listagem 5. Classe ClienteController

```
TClienteController = class(TInterfacedObject, IController)
private
  FModel: TCliente;
  class var Finstance: TClienteController;
  constructor PrivateCreate;
protected

public
  class function GetInstance: TClienteController;
  constructor Create;
  destructor Destroy; override;
  procedure Save;
  function Find(ID: String): TBaseObject;
  function New: TBaseObject;
published
  property Model: TCliente read FModel;
end;

implementation

uses
  SysUtils;

constructor TClienteController.PrivateCreate;
begin
  inherited Create;
end;

constructor TClienteController.Create;
begin
  raise Exception.Create('Para Obter um ClienteController Invoque o       Metodo
GetIntance');
end;

destructor TClienteController.Destroy;
begin
  inherited;
end;

function TClienteController.Find(ID:String): TBaseObject;
begin
  Result := FModel.Buscar(ID);
end;

class function TClienteController.GetInstance: TClienteController;
begin
  if not Assigned(Finstance) then
    Finstance := TClienteController.PrivateCreate;
  Result := Finstance;
end;

function TClienteController.New: TBaseObject;
begin
  FModel := TCliente.Create;
  Result := FModel;
end;

procedure TClienteController.Save;
begin
  FModel.Salvar;
end;

end.
```

risco dela já estar implementando uma interface como ocorre no caso da herança. Veja na terceira linha que apenas adicionamos o objeto passado como na lista de observadores.

Na seqüência temos a implementação do construtor da nossa classe. Nele instanciamos a classe *TInterfaceList* que irá funcionar como repositório de *observadores* e em seguida geramos uma nova *GUID* e associamos a propriedade *ID* do nosso objeto, assim garantimos um identificador único para cada objeto que for criado em nosso modelo de dados.

O código do método *Desattach* apenas remove o objeto referido como parâmetro da lista de notificações, assim este objeto não será mais notificado. No *destructor* apenas destruímos o objeto *TInterfaceList* para, por fim, chegarmos ao tão aguardado método *Notify*. Este será o método chamado toda fez que o estado no objeto em questão for alterado, ou seja, ao menor sinal de iteração com o objeto o método *Notify* deverá ser invocado e tem um motivo óbvio para isso.

Observe na *Listagem 3* que nosso código começa fazendo um loop com o método for in em nossa lista de observadores e para cada item da nossa lista invocamos o método *update* e passamos o próprio objeto como parâmetro para que o observador saiba por quem ele foi notificado. Assim cabe agora ao observador, que no MVC será a *View*, tomar a providência necessária para atualizar os dados do modelo na *View*. E é isso que veremos mais a frente.

Com nossa classe base pronta vamos a criação da nossa classe de negócio que será a classe *TCliente* que herdará de nossa *BaseObejct* já "nascendo" com a possibilidade de ser observada por um observador. Assim ainda em nosso pro-

Nota do DevMan

TGUID é um record no Delphi utilizado para representar um GUID (Global Unique Identifier). A declaração de uma interface pode conter um GUID que é declarado como uma cadeia de caracteres entre colchetes da seguinte forma: ['{xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxx}'] onde cada x é um digito hexadecimal 0 – F. Você pode gerar suas GUID's no Delphi pressionando Ctrl + Shift + G no Code Editor.

jeto no Delphi adicione mais uma *Unit*, salve-a como *uCliente.pas* e nela declare o código da **Listagem 4**.

O código da **Listagem 4** mostra a nossa classe cliente ainda sem a implementação dos métodos e *property*. Para gerar a implementação basta pressionar *Ctrl + Shift + C*. O código gerado nada mais é do que a implementação das *property* o que dispensa comentário e mais a implementação dos métodos *Salvar, Deletar* e *Buscar*. Esses métodos são apenas ilustrativos e estão ai para demonstrar a mudança de estado do objeto, e como estamos falando sobre MVC a cada alteração no estado do objeto os observadores, nesse caso serão as *views*, deverão ser notificadas.

O método salvar, quando chamado, deverá persistir o objeto em questão no banco de dados. Não á escopo deste artigo tratar de persistência, sendo assim além de invocar o método *notify* na implementação do método salvar você invocaria sua classe de persistência e solicitaria a ela que persistisse seu objeto. Já o método *Buscar* servirá para carregar na classe informações de um objeto específico. Como em OO temos o conceito de *Object ID*, ou seja, cada objeto em seu modelo deverá possuir um identificador único, passamos apenas o ID do objeto para que possa ser carregado no mesmo os dados referente ao objeto possuidor do ID. Como dito acima, não teremos aqui a implementação do método como um todo, pois isso também seria função da classe de persistência, porém o método *notify* será invocado para notificar os observadores que os dados do modelo foram alterados. Sendo assim na implementação deste dois métodos apenas invoque o método *notify*.

Com isso concluímos nosso *Model*. O mesmo já pode ser "persistido" e notificado aos observadores caso seu estado seja alterado. Passemos então a criação da nossa classe de controle.

Controller

Ainda neste mesmo projeto, adicione uma nova *Unit* e salve-a como *ClienteController.pas*. É nesta *Unit* que iremos criar a classe de controle para iterar com nosso modelo. Assim, esta classe de controle esta apresentada na **Listagem 5**.

Se você atentou a definição do padrão MVC e a imagem da **Figura 3** irá perceber que a função do *Controller* é exatamente mapear as ações do usuário na *View* e ter inteligência suficiente para saber qual método invocar no modelo. Por isso é muito comum que você encontre no *controller* os métodos de *CRUD*, para que o usuário através da *View* possa invocar métodos para manipular um determinado *Model*. Assim começamos a declaração do nosso *Controller* implementando a interface *IController* e desta maneira temos em *Controller* os métodos presentes na interface *IController*. Já podemos então permitir a *View*, através do *Controller*, criar um novo Objeto, Buscar um Objeto e salvar alterações no modelo. Porém vamos atentar para o fato de, se o *Controller* tem a função que tem, não justifica ter num projeto dois *ClienteController* se o usuário só poderá manipular um cliente por vez (pelo menos essa e a minha abordagem).

Assim vamos aplicar a nossa classe *ClienteController* o padrão de projeto *Singleton*. Temos os métodos necessários para poder implementar o *Singleton*. Uma variável de classe para guardar a instância quando criada, um método de classe (*GetInstance*) para retornar uma instância de *TClienteContoller* e um construtor privado para construir nosso objeto.

Esse construtor é privado, pois o construtor público foi invalidado, observe na linha 26. Se alguém tentar instanciar um objeto do tipo *TClienteController* irá receber um erro. Desta forma garantimos que só teremos um único objeto instanciado em nossa aplicação, e como eu posso garantir isso? Simples. Como a única maneira de instanciar um Objeto *TClienteController* é através do método *GetInstance*, então é nele que fazemos a verificação para saber se já não há um objeto instanciado. Caso não haja um objeto instanciado então criamos um e o armazenamos na variavél *FInstance*. Da próxima vez que o método *GetInstance* for chamado não será criado um novo objeto apenas retornado o objeto que já existe.

Há apenas uma ressalva aqui. Como disse no início do artigo que este exemplo poderia ser feito com versões do Delphi posteriores a 3, então aqueles que estiverem com versões anteriores a 2006 deverão substituir a *class var* por uma variável global, pois versões anteriores a 2006 não suportam *class var*.

Continuando no código da **Listagem 5** temos os métodos que mapearão a ação do usuário na *View* para o *Model*. Assim fica simples entender o que será feito na implementação de cada método. Mais abaixo é mostrado a implementação do método *Save*, ele invoca o método salvar no modelo, que se encarrega que "persistir" os dados (neste artigo não abordamos isso) e notificar a *View*. Assim quando o usuário clicar no botão salvar o modelo mudará de estado e a *View* receberá a notificação e se atualizará. Isso será feito adiante.

Em seguida faz a mesma coisa porém com o método *Buscar*. Observe que sempre fazemos a chamada de *FModel* que é a propriedade do próprio *Controller*. porém uma propriedade somente leitura. Isso para garantir que ninguém ira passar um *Model* para o *Controller* garantindo assim que a única maneira de um *Model* ser associado ao *Controller* e através do método *New*, e é isto que mostra nas seguintes.

Cliente View

Vamos adicionar ao nosso projeto mais um formulário. Salve-o como *uFrmCliente.pas* e renomeie-o para *FrmCliente*. Relembrando, a função da *View* é permitir que o usuário interaja com o modelo com suas ações sendo mapeadas pelo *Controller*. Sendo assim temos que ter em nossa *View* controles visuais para podermos valorar nosso *Model*.

O principal será feito agora. Como no padrão MVC a *View* observa o modelo

Nota do DevMan

O padrão de projeto Singleton assegura que haja somente uma instância de uma classe, e fornece um ponto de acesso global para acessá-la [GoF]. Utilizado em situações onde queremos garantir que só haverá um e somente um objeto deste tipo na aplicação. Isto é feito invalidando o construtor da classe e criando um método de classe que retorne uma instancia deste objeto.

Listagem 6. Implementação da interface IObserver

```
TFrmCliente = class(TForm, IObserver)
  private
    FController: TClienteController;
  public
    constructor Create(AOwner: TComponent; Controller: TClienteController);reintroduce;
    procedure Update(Observable: IObservable);
  end;
```

Listagem 7. Implementação dos Métodos da View

```
constructor TFrmCliente.Create(AOwner: TComponent;
  Controller: TClienteController);
begin
  inherited Create(AOwner);
  FController := Controller;
end;
procedure TFrmCliente.Update(Observable: IObservable);
begin
  EdtID.Text       := FController.Model.ID;
  EdtNome.Text     := FController.Model.Nome;
  EdtEmail.Text    := FController.Model.Email;
  EdtTelefone.Text := FController.Model.Telefone;
end;
```

Listagem 8. Implementação dos métodos

```
procedure TFrmCliente.BtnNovoClick(Sender: TObject);
begin
  FController.New;
  FController.Model.Atach(Self);
end;
procedure TFrmCliente.BtnPesquisarClick(Sender: TObject);
begin
  FController.Model.Buscar('');
end;
procedure TFrmCliente.BtnSalvarClick(Sender: TObject);
begin
  FController.Model.Nome := EdtNome.Text;
  FController.Model.Email := EdtEmail.Text;
  FController.Model.Telefone := EdtTelefone.Text;
  FController.Save;
end;
```

Listagem 9. Chamando a View do Cliente

```
procedure TForm1.BtnClienteClick(Sender: TObject);
var
  C: TClienteController;
begin
  C := TClienteController.GetInstance;
  FrmCliente := TFrmCliente.Create(Self,C);
  FrmCliente.Show;
end;
```

então temos que transformar nossa *View* num *observador* e faremos isso implementando a interface *IObserver* na nossa classe *TFrmCliente*. Desta forma o formulário de cliente é obrigado a implementar o método *Update*, que é extremamente importante, pois é este método que será responsável por atualizar a *View*. Observe a **Listagem 6**, ela mostra a como fica a declaração da classe cliente com a interface implementada e os demais métodos requeridos.

Em primeiro lugar é notório a presença do método *update*, isso porque implementamos a interface *IObeserver*. Com isso nosso *form*, ou melhor, *View* tem que poder ser atualizado com o método *update*. Lembre-se que em nosso *Model* a cada mudança de estado invocamos o método *Notify* e este por sua vez faz um loop na lista de observadores e chama para cada um o método *update*. Ainda no código da **Listagem 6** temos uma atributo do tipo *TClienteController*, isso para podermos guardar a referência do *Controller* que a *View* está se comunicando. Toda *View* tem suas ações, ou melhor, ações do usuário mapeadas para um *Controller* daí o motivo de ainda na **Listagem 6** nos reescrevermos o construtor do nosso formulário. Observe que o construtor agora pede dois parâmetros: o *owner* que já é padrão para todo componente e agora também pede um *Controller*, assim garantiremos que ao criar uma *View* já teremos um *Controller* associado a ela. E para que isto?

Tenha sempre em mente que as ações do usuário mapeadas pela *View* devem ser enviadas ao *Model* através do *Controller*, por isso esta condição de só criarmos uma *View* com um *Controller* previamente criado. A implementação dos métodos explica-se por si só. Observe a **Listagem 7**.

Veja que no construtor da nossa *View* nós invocamos o construtor de *TForm* e passamos para ele o *Owner* passado como parâmetro. O segundo parâmetro, o *Controller*, nós guardamos no *Field FController*, com isso garantimos que esta *View* o será criada se um *Controller* for passado como parâmetro. Na seqüência vemos a implementação do método *Update*. Observe que a *View* ao ser notificada atualiza os controles da *View* repassando para os mesmos os novos dados do modelo. Note bem que os valores são obtido via *Controller*, porém também poderia ser feito fazendo um *typecasting* de *IObservable* para *TCliente*.

Para concluir nossa *View* vamos a implementação dos eventos clique dos botões novo, pesquisar e salvar. A **Listagem 8** se encarrega de exibir os métodos.

Como dito anteriormente o *Controller* mapeia as ações executadas na *View* para o *Model*. E a implementação dos métodos na **Listagem 8** ilustra bem isto. O clique do botão novo invoca o método *new* do *Controller* e este por sua vez cria um novo objeto *Cliente* e o armazena internamente na propriedade *Model*. Feito isso o próximo passo é inscrever esta *View* onde estamos neste novo modelo para que a mesma possa receber as notificações necessárias ao longo do ciclo de vida o objeto. O botão pesquisar apenas invoca o método Buscar do modelo que (se tivesse implementação com DAO) configuraria as propriedades do Objeto Cliente com os dados oriundos do banco de dados.

Por fim temos o código do clique do botão salvar. Este por sua vez pega os valores dos *edits*, atribui as propriedades do *Model* que esta sendo controlado pelo *Controller* no momento e invoca o método *Save*. Este método e mapeado para o *Model* que se encarrega de persistir o objeto e notificar os Observadores.

Com isto temos nosso MVC implemen-

tado e pronto para uso. Vamos, em caráter de teste, invocar nossa *View* e realizar os testes. Assim no *form* principal da aplicação coloque um botão para invocar a *View Cliente* e no evento *OnClick* insira o código da **Listagem 9**.

No clique do botão que invocamos a *View* do cliente nós declaramos uma variável do tipo *TClienteController* e *Pedimos* um *Controller* para ela, lembrando que o *ClienteController* é um *Singleton* e como tal só termos um instanciado por vez em nossa aplicação. Em seguida criamos nossa *View* do cliente e repare que ao chamar o construtor ele nos pede além do *owner* um *Controller*, e é ai que passamos a variável C com o *Controller* criado acima. Feito isso exibimos a *View* com o já conhecido método *Show*. Rode a aplicação, clique no botão novo e em seguida no botão salvar (**Figura 7**).

Veja que nosso MVC já esta em pleno funcionamento. Ao clicar em novo o *Controller* criou um novo objeto e neste momento já temos um *GUID* associado a e,le porém como não ouve notificação a *View* não esta sabendo. Quando chamamos o método salvar a notificação é feita a todos os observadores e como nossa *View* está inscrita na lista a mesma é notificada e os dados atualizados na *View*. E óbvio que uma aplicação *Stand Alone* como esta pode não mostrar todo o potencial deste padrão arquitetural, mas experimente chamar outro formulário e clique em novo. Depois altere os dados no primeiro *Form* e salve. Você irá perceber que o segundo *Form* também será atualizado com os dados do primeiro. A **Figura 8** mostra o exemplo citado em funcionamento.

Conclusão

O padrão MVC é muito mais comum na WEB, isso porque fica fácil criar um repositório de *Controllers*, que ficaria na sessão e assim conseguiria notificar todas as *Views*. Porém nada nos impede de aplicarmos em aplicações Win32. Porém há de se ficar claro que não justifica aplicar o padrão MVC em uma aplicação *Stand Alone* como fizemos aqui. Mas imagine este padrão aplicado numa Aplicação que tenha um servidor de objetos, COM+ por exemplo. Poderíamos ali criar nosso repositório de objetos e um *Controller Singleton* para mapear as ações dos usuários. O importante é deixar claro que este é um padrão arquitetural que pode ser, e deve ser aplicado junto com outros padrões de projeto, pois como os outros se trata também de uma boa prática de programação.

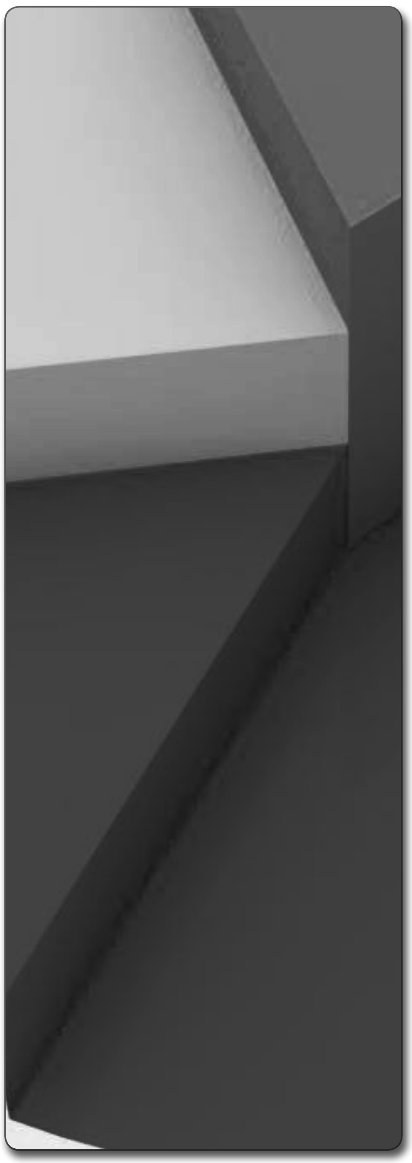
Eu fico por aqui, espero que tenha contribuído com mais este artigo para o nosso aprendizado. Até a persistência. ●

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback



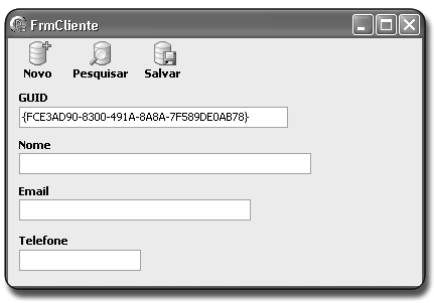


Figura 7. Novo cliente criado e salvo

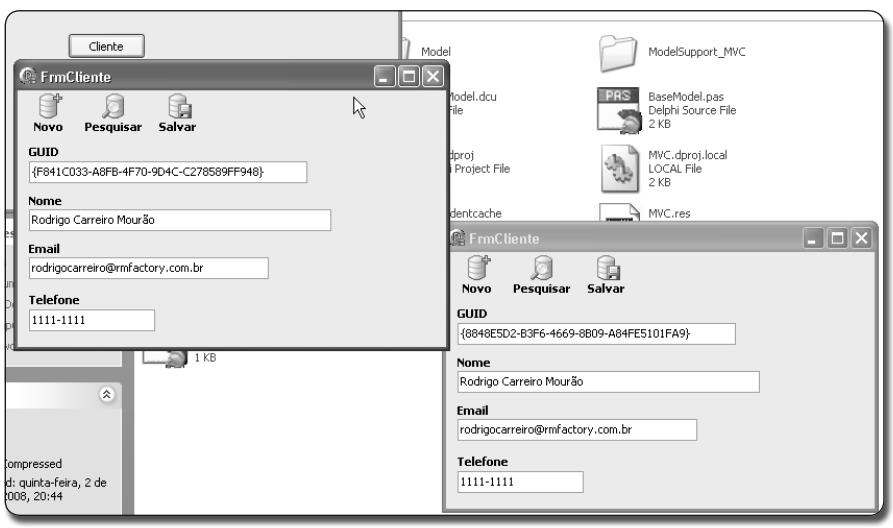
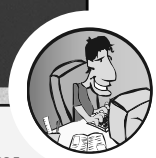


Figura 9. Duas View sendo atualizadas ao mesmo tempo

Nesta seção você encontra artigos sobre a linguagem PHP e a ferramenta Delphi for PHP

Relatórios no PHP com arquivos PDF

Aprenda a gerar arquivos PDF no PHP para emissão de relatórios



Resumo DevMan

Todos nós sabemos que grande parte dos sistemas de como um todo acabam não tenho tanta utilidade quando não há relatórios. É assim com aplicações Win32, mais cedo ou mais tarde o cliente pede para desenvolver algum tipo de relatório, tais como: comissão de vendas, vendas no mês, controle de estoque, gráfico de vendas, fluxo de caixa etc.

É possível desenvolver relatórios em Delphi for Win32 numa facilidade incrível, já que temos diversos editores de relatórios para todo gosto e bolso.

Na web não é diferente, os relatórios serão solicitados e teremos que estar preparados para desenvolver a solução para o cliente. A diferença maior entre aplicações Win32 e Web é que em Web a gama de opções para o desenvolvimento é bastante pequena. Por isso, nesse artigo quero mostrar uma boa solução para desenvolver relatórios na Web utilizando a biblioteca FPDF que é específica para programação PHP.

Nesse artigo veremos

- Como utilizar a classe FPDF para criar arquivos PDF;
- Como gerar relatórios em PDF para Web Sites PHP.

Qual a finalidade

- Implementar uma solução para relatórios, em formato profissional, no PHP.

Quais situações utilizam esses recursos?

- Desenvolvimento de sistemas e sites com PHP onde se necessita de relatórios para obter-se informações.

Fabrício Desbessel

(fabricio@fabricio.pro.br)

é professor de Linguagem de Programação do Curso Técnico em Informática do Centro Tecnológico Frederico Jorge Logemann de Horizontina/RS e da FAHOR Faculdade Horizontina. Delphiano de coração está sempre disposto a provar que com o Delphi sempre teremos a melhor solução, até mesmo com o PHP. Site www.fabricio.pro.br.

Quem tem uma aplicação *WEB* sabe que haverá solicitação de relatórios que deverão ter uma aparência profissional, que não percam a formatação de uma impressora para outra e, o mais importante, possam ser salvos para arquivo ou enviados via *e-mail*.

O formato ideal para relatórios na *Web* atualmente é o *PDF*, pois além de imprimir nas mais variadas impressoras sem perder formatação, podemos guardar os arquivos e enviá-los por e-mail ou qualquer outra mídia.

Portanto, nesse artigo criaremos arquivos *PDF* utilizando a biblioteca *PDFLib* do *PHP*. Vamos aprender na prática!

Preparando a utilização

Como o *FPDF* é uma classe, basta fazer o *download* do mesmo e colocar em um diretório dentro da sua aplicação. Para baixar acesse o *site www.fpdf.org* e o *link Downloads*. Escolha a versão mais atual

até o fechamento dessa edição. Depois de fazer o *download* descompacte o arquivo. Como resultado, tem-se o arquivo da classe (*fpdf.php*) que deverá ser copiada para o diretório da sua aplicação, bem como a pasta *font* que contém as fontes *TrueType* que poderão ser utilizadas.

Os demais arquivos são a documentação, um tutorial e informações. No *site* também existe um tutorial traduzido para o português do Brasil que pode ser baixado.

FPDF é liberado sob uma licença permissiva: não há qualquer restrição de utilização. Você pode inserí-lo livremente na sua aplicação (comercial ou não), com ou sem modificações.

Listagem 1. Relatório Olá Mundo!

```
<?php
  define('FPDF_FONTPATH','C:/CD/PDF/fpdf16/font/');
  require('fpdf16/fpdf.php');
  $pdf=new FPDF();
  $pdf->AddPage();
  $pdf->SetFont('Arial','B',50);
  $pdf->Cell(40,10,'Olá Mundo!');
  $pdf->Output();
?>
```

Listagem 2. Relatório com várias linhas seguidas

```
<?php
  define('FPDF_FONTPATH','C:/CD/PDF/fpdf16/font/');
  require('fpdf16/fpdf.php');
  $pdf=new FPDF('P','mm','A4');
  $pdf->AddPage();
  $pdf->SetFont('Arial','',10);
  $pdf->Cell(190,4,'Olá Mundo!',1,1);
  $pdf->Cell(190,4,'Proxima Linha 1',1,1);
  $pdf->Cell(190,4,'Proxima Linha 2',1,1,'C');
  $pdf->Cell(190,4,'Proxima Linha 3 - Texto que será alinhado à direita',1,1,'R');
  $pdf->Output();
?>
```

Conhecendo alguns recursos disponíveis

Para geração de arquivos *PDF*, a *FPDF* oferece suporte para inserir imagens nos formatos *JPEG*, *PNG* e *GIF*. Para a edição do texto, qualquer fonte *TrueType* pode ser utilizada, desde que seja distribuída juntamente com a aplicação, podendo-se também definir a cor das fontes. Quanto à formatação do texto, é possível criar cabeçalhos e rodapés, criar *links* hipertexto, sendo também possível a quebra automática de linhas e páginas. Além de também definir as medidas das margens.

> **Nota**: Para inserir no relatórios imagens do tipo *Gif*, será necessário ter carregada no *PHP* a biblioteca GD. Em instalações do *Delphi for PHP* essa biblioteca já vem carregada. Se não seu caso, verifique o arquivo *PHP.ini*.

E o melhor de tudo é que o arquivo *PDF* resultante estará devidamente compactado, o que torna sua exibição mais rápida, em um tempo aceitável para aplicações *Web*.

Construindo os primeiros relatórios

Vamos iniciar nosso primeiro teste criando uma aplicação bastante conhecida de todos, a famosa e tradicional mensagem: *Ola Mundo!* Abra seu editor preferido e digite o código da **Listagem 1**. Minha preferência é pelo *Delphi For PHP,* para agilizar a execução dos testes e ter a possibilidade de *debug* do código *PHP,* mas sinta-se a vontade para utilizar seu software de preferência, como o *Bloco de Notas* do *Windows*, por exemplo.

A primeira linha de comando no código da **Listagem 1** serve para definir o valor de uma constante, utilizada pela classe para saber o diretório onde estão as fontes Perceba que no meu caso apontei para *C:/CD/PDF/fpdf16/font/* , ou seja, local. Obviamente quando distribuir essa aplicação terei que modificar o caminho para que se adapte as *urls* de meu sistema *on-line*. Mais abaixo vemos uma chamada ao método *SetFont*, que serve para escolher a fonte, sua formatação e tamanho. O comando *require* do *PHP* inclui o arquivo passado como parâmetro no *script* atual, exigindo sua existência. Essa inclusão é necessária para adicionar o código da classe *FPDF* ao nosso *script PHP*.

Para iniciar a criação de um relatório declara-se uma variável que irá receber um objeto da classe *FPDF*. Esse objeto representa o arquivo *PDF* que vamos criar. E ele adicionamos uma página através do método *AddPage* e, para escrever utiliza-se o método *Cell* que tem como parâmetro os tamanhos *width*, *heigth* e o texto a ser impresso.

O *Cell* na verdade define uma célula com largura (*width*) e altura (*heigth*) onde pode-se, inclusive, centralizar ou alinhar o texto na direita e na esquerda. Para exibir o relatório criado, utiliza-se o método *Output* que faz uma saída redirecionando a página para um arquivo *PDF* que deverá ser carregado no *browser*.

Agora vamos evoluir um pouco o relatório codificando conforme a **Listagem 2**, onde acrescentamos alguns parâmetros nos métodos.

Na **Listagem 2** criamos o relatório e já configuramos a orientação e formato da folha. Isso é feito na linha

```
$pdf=new FPDF('P','mm','A4');
```

O primeiro parâmetro (P) está definindo a orientação *Portrait* (retrato). Se você deseja fazer um relatório no formato horizontal, defina o parâmetro como (L) que significa *Landscape* (paisagem). Também pode-se definir a unidade de medida do relatório, esse é o segundo parâmetro. No caso, foi definido em milímetros (mm), mas pode-se utilizar: *(cm) centímetros*, *(pt) pontos* e *(in) polegadas*. Essa definição de unidade de medida será aplicada no documento todo e será utilizada para qualquer posicionamento dentro do mesmo. Por isso é muito importante definir a unidade padrão antes de organizar os relatórios, pois mudar depois poderá desorganizar completamente a formatação. Finalizando os parâmetros do relatório pode-se definir o formato da folha e os valores que podem ser utilizados são: *A3*, *A4*, *A5*, *Letter* e *Legal*. Lembrando que no Brasil o formato padrão é o *A4*.

O método *AddPage* também possui o parâmetro de orientação e formato da página, o que possibilita que você tenha páginas diferenciadas dentro do mesmo relatório.

Listagem 3. Relatório com cabeçalho e rodapé

```
<?php
  define('FPDF_FONTPATH','C:/CD/PDF/fpdf16/font/');
  require('fpdf16/fpdf.php');

class RELATORIO extends FPDF{
  function Header(){
    $this->Image('logo_CD.gif',10,8);
    $this->SetFont('Arial','B',15);
    $this->Cell(80);
    $this->Cell(100,10,'Relatório de Demonstração do FPDF',0,0,'C');
    $this->Ln(20);
  }
  function Footer(){
    $this->SetY(-15);
    $this->SetFont('Arial','I',8);
    $this->Cell(0,10,'Page '.$this->PageNo().'/{nb}',0,0,'C');
  }
}
$pdf=new RELATORIO();
$pdf->AliasNbPages();
$pdf->AddPage();
$pdf->SetFont('Times','',12);
for($i=1;$i<=40;$i++)
  $pdf->Cell(0,10,'Texto da linha número '.$i,0,1);
$pdf->Output();
?>
```

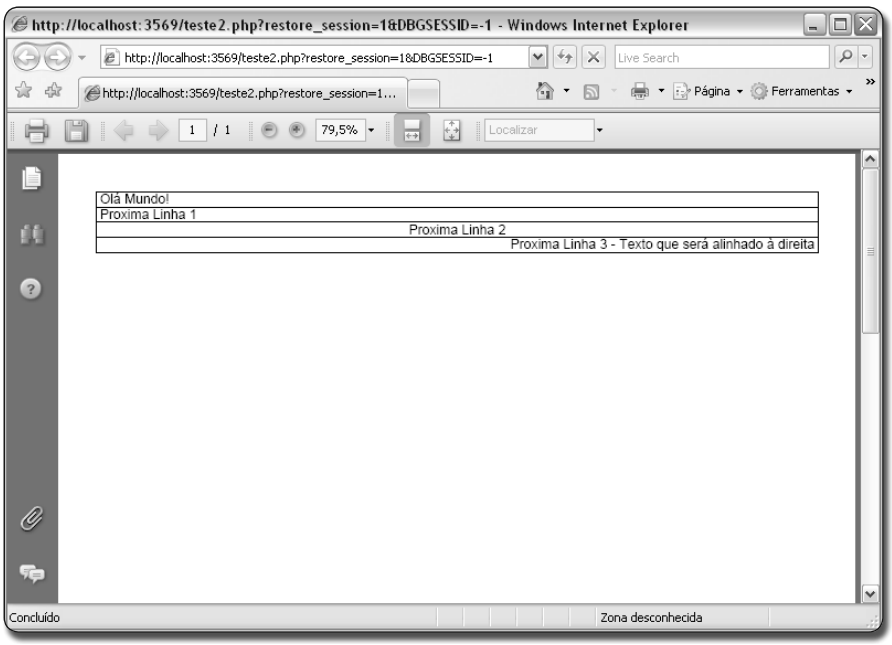


Figura 1. Relatório criado através do script da Listagem 2

Para definir a fonte, utilizou-se o método *SetFont* com o parâmetro de fonte como *Arial*. O parâmetro seguinte está em branco, ou seja, sem nenhuma formatação original, lembrando que é possível formatar como negrito (B), itálico (I) e sublinhado (U). Inclusive pode-se formatar a fonte com todas as opções passando no segundo parâmetro *BIU*. No final do *SetFont* tem-se o tamanho da fonte, em pontos, que, se não for informado será 12 (doze). Ao definir um *SetFont*, essa definição valerá para todo o relatório até que exista outra definição, ou seja, outro *SetFont*.

Nas linhas que chamam o método *Cell* na **Listagem 2,** definiu-se a largura (*width*) em 190, sendo que uma folha do tipo A4 tem 210 mm, deixando 20 mm de margem, ou seja, 2 centímetros. Como altura utilizou-se a medida de 4 mm o que é suficiente para um texto com fonte *Arial* tamanho 10. Ainda no método *Cell*, após o texto que deverá ser escrito colocou-se no próximo parâmetro o número '1' que significa que uma borda será aplicada em toda a célula, o que é bem interessante em relatórios que precisam de linhas a cada registro. Depois da borda, tem-se mais um parâmetro com '1' para definir uma quebra de linha após o texto. Com isso a próxima chamada ao método *Cell* estará posicionado na linha seguinte.

Na chamada escreve o texto *Próxima Linha 2* tem-se, como último parâmetro a letra ***C*** que define que o texto deve ser centralizado dentro da célula, levando em conta seu tamanho (largura). Pode-se ainda definir a posição do texto com *R* (direita) e *L* (esquerda), como demonstra-se na última chamada ao método *Cell* que alinha o texto na margem direita. Colocando esse *script* para executar tem-se como resultado o exibido na **Figura 1**.

Adicionando cabeçalho e rodapé ao relatório

A classe *FPDF* possui dois métodos que não foram implementados, deixando essa possibilidade de extensão dos mesmos. Esses métodos são o *Header* (cabeçalho) e o *Footer* (rodapé). Portanto, para criar um cabeçalho e rodapé faze-se uma extensão da classe definindo esses dois métodos. É importante salientar a necessidade de utilizar o *$this* para chamar os métodos da própria classe. Veja na **Listagem 3** um relatório com cabeçalho e rodapé, bem como recursos de imagem e contador de páginas.

No código da **Listagem 3** estende-se (herança) a classe *FPDF* criando a classe *Relatorio*. Com essa extensão podemos definir os métodos *Header* e *Footer*.

No *Header*, invocamos o método *Image* para adicionar uma imagem contida no mesmo diretório do *script PHP*, que será iniciada na posição horizontal 10 e vertical 8. Tem-se a opção ainda de passar mais um parâmetro, o tamanho que a imagem deve ter de largura, assim ela será redimensionada proporcionalmente. No caso da **Listagem 3** não declarou-se o tamanho da imagem e a mesma será exibida no tamanho original. Na seqüência tem-se uma chamada ao *Cell* passando como parâmetro 80 (oitenta), com o objetivo de criar uma célula de espaço entre a figura e o título do relatório que é impresso através da próxima chamada. No final do *Header* tem-se uma chamada ao método *Ln* que cria uma linha em branco com uma

quebra, onde seu parâmetro é a altura que se não for informada será a mesma altura da última chamada ao *Cell*.

No *Footer* vemos uma chamada a *SetY* que serve para posicionar a escrita em relação vertical. O valor negativo refere-se à base do relatório, ou seja, tem-se 15 mm de baixo para cima. Depois imprime-se o número da página atual, através do método *PageNo*, bem como a quantidade de páginas através do *alias {nb}* que é resultado da função *AliasNbPages*. Assim fecha-se a extensão da classe *FPDF*.

Continuando, tem-se a criação do relatório que agora é uma variável do tipo *Relatorio* que é a classe estendida e que contém o cabeçalho e rodapé. Faz-se necessário chamar o método *AliasNbPages* para definir o alias que conterá o número final de páginas do relatório. Define-se uma fonte do tipo *Times* e tem-se um *looping* para imprimir o mesmo texto várias vezes, ocasionando quebras de páginas. Na **Figura 2** apresenta-se o resultado do relatório da **Listagem 3**.

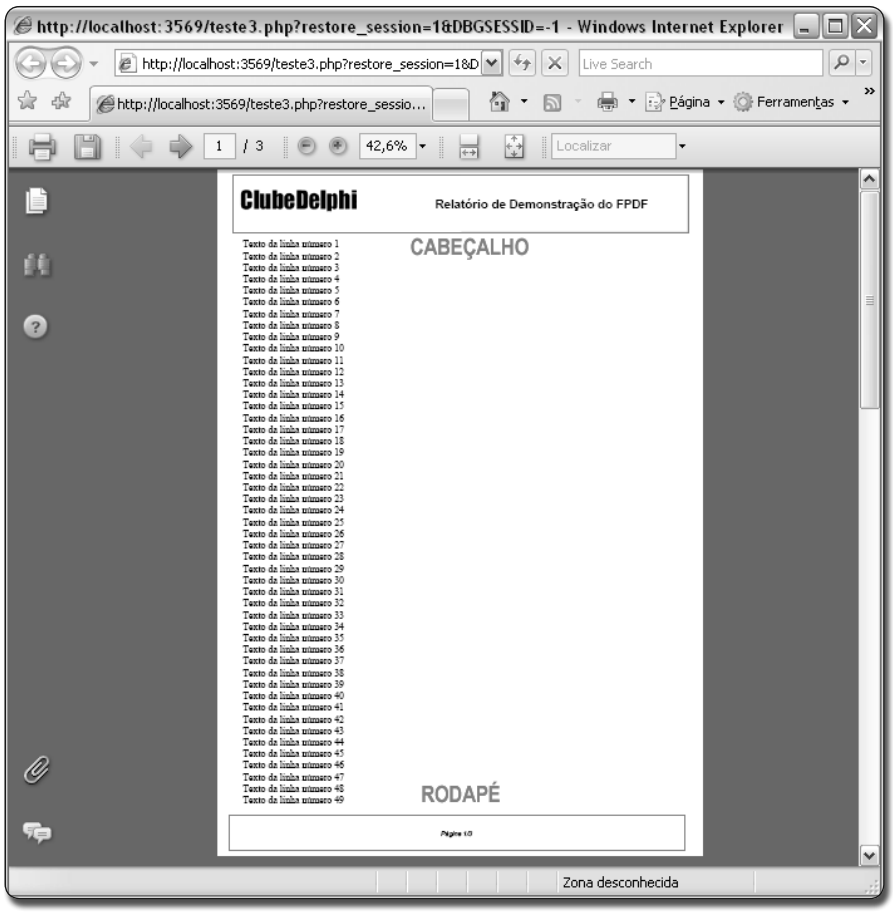


Figura 2. Relatório com cabeçalho e rodapé

Colorindo o relatório

Para se ter uma idéia do quão interessante é a classe *FPDF*, temos um recurso interessante. Mesmo o relatório sendo *PDF* é interessante utilizar cores para destacar informações importantes e que devem ser rapidamente identificadas. A classe possui isso e veremos que é bem fácil colorir um relatório. Indo direto ao ponto tem-se na **Listagem 4** a criação de um relatório com cores.

Listagem 4. Relatório com cores

```
<?php
define('FPDF_FONTPATH','C:/CD/PDF/fpdf16/font/');
require('fpdf16/fpdf.php');
$pdf=new FPDF();
$texto='Teste com Cores';
$pdf->SetFont('Arial','',14);
$pdf->AddPage();
$w=$pdf->GetStringWidth($texto)+6;
$pdf->SetX((210-$w)/2);
$pdf->SetDrawColor(0,80,180);
$pdf->SetFillColor(230,230,0);
$pdf->SetTextColor(220,50,50);
$pdf->SetLineWidth(1);
$pdf->Cell($w,9,$texto,1,1,'C',true);
$pdf->Ln(10);
$pdf->Output();
?>
```

Na **Listagem 4,** criamos uma caixa personalizada e nela escrevemos um texto. Utilizamos o método *GetStringWidth* para descobrir o tamanho que o texto contido na variável terá no relatório, levando em conta a unidade de media do mesmo. A esse resultado soma-se mais 6 (seis) para ter um tamanho ideal da caixa que será criada. Utilizou-se o método *SetX* para posicionar o texto em relação horizontal, fazendo uma conta para posicionar o resultado bem ao centro. Essa conta é simples, pegou-se o tamanho do relatório (210), diminuiu-se o tamanho do texto ($w) e dividiu-se por dois (2).

O *SetDrawColor* é utilizado para definir a cor das linhas (bordas) que compõem a célula, utilizando como parâmetro o padrão de cores *RGB* (*Red*, *Green*, *Blue*), onde define-se a quantidade de vermelho (0 a 255), a quantidade de verde (0 a 255) e a quantidade de azul (0 a 255). Para mudar a cor interna da célula utiliza-se o *SetFillColor* que também utiliza o padrão *RGB*, bem como o método *SetTextColor* que define a cor do texto propriamente. Ainda pode-se definir o tamanho da linha da célula através do método *SetLineWidth*.

Nota do DevMan

No padrão *RGB* todas as cores são formadas por misturas de vermelho, verde e azul, sendo assim cada cor possui uma representação numérica que informa a quantidade de cada mistura. Por exemplo, a cor branca é representada como *RGB*(1,1,1). Para obter esses valores você pode utilizar programas de edição gráfica, como o *Photoshop*.

Listagem 5. Listagem de dados em formato de tabela

```
<?php
  define('FPDF_FONTPATH','C:/CD/PDF/fpdf16/font/');
  require('fpdf16/fpdf.php');
  $MYSQL = mysql_pconnect("localhost","root","root");
  mysql_select_db('direto',$MYSQL);
  $SQL = "select * from clientes order by nome";
  $MResultado = mysql_query($SQL, $MYSQL);
  class RELATORIO extends FPDF{
    function Header(){
      $this->Image('logo_CD.gif',10,8);
      $this->SetFont('Arial','B',15);
      $this->Cell(80);
      $this->Cell(100,10,'Listagem de Clientes',0,0,'C');
      $this->Ln(20);
    }
    function Footer(){
      $this->SetY(-15);
      $this->SetFont('Arial','I',8);
      $this->Cell(0,10,'Página '.$this->PageNo().'/{nb}',0,0,'C');
    }
  }
$pdf=new RELATORIO();
$pdf->AliasNbPages();
$pdf->AddPage();
$pdf->SetFont('Times','B',12);
//Colunas
$pdf->Cell(20,4,'Código',1,0,'C');
$pdf->Cell(60,4,'Nome',1,0,'C');
$pdf->Cell(30,4,'D.Nasc',1,0,'C');
$pdf->Cell(30,4,'Fone',1,0,'C');
$pdf->Cell(50,4,'E-mail',1,1,'C');
$pdf->SetFont('Times','',12);
//Registros
while ($MColuna = mysql_fetch_assoc($MResultado)){
  $pdf->Cell(20,4,$MColuna["id"],1,0,'C');
  $pdf->Cell(60,4,$MColuna["nome"],1,0);
  $pdf->Cell(30,4,$MColuna["datanasc"],1,0,'C');
  $pdf->Cell(30,4,$MColuna["fone"],1,0,'C');
  $pdf->Cell(50,4,$MColuna["email"],1,1);
}
mysql_free_result($MResultado);
$pdf->Output();
?>
```

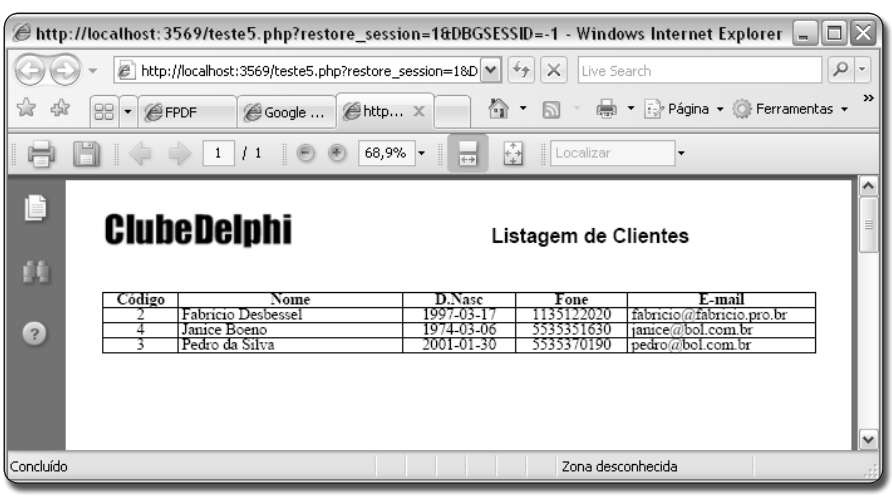


Figura 3. Relatório com dados no formato de tabela

Listando dados no formato de tabelas

Até agora, vimos como fazer a impressão de relatórios simples, sem acesso a dados. Agora vamos à parte que mais interessa: uma listagem de dados. Com tudo que já foi escrito nesse artigo fica fácil pensar e fazer uma listagem que coloca, no formato de tabela, o resultado de uma consulta ao banco de dados. Mas, para facilitar vamos ver um exemplo na **Listagem 5,** que busca dados de uma tabela de *Clientes* em um banco *MySQL*.

Na **Listagem 5,** tem-se uma consulta ao banco de dados *MySQL*, em uma tabela de clientes. Montou-se o relatório com cabeçalho e rodapé, como visto anteriormente. Na questão da tabela criaram-se os títulos da coluna de forma centralizada e definindo um tamanho. Na última coluna (*E-mail*) colocou-se 1 no parâmetro que quebra a linha para que o registro seja iniciado abaixo dos títulos.

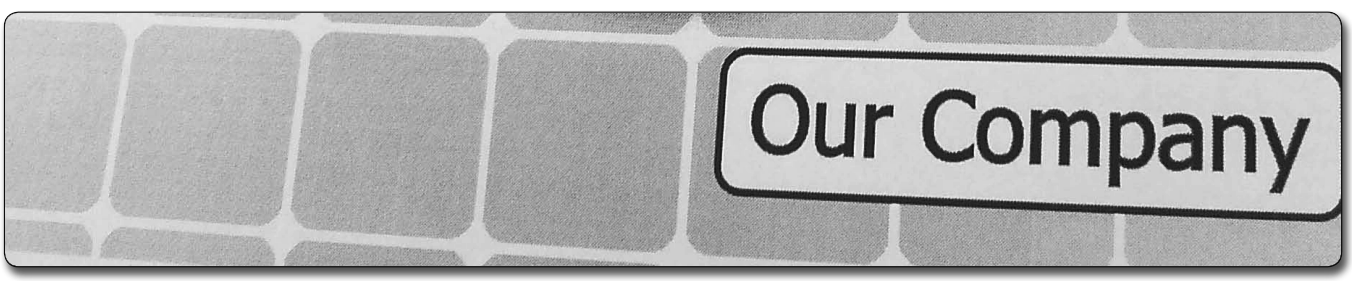
Com o *while* criou-se um *loop* até o final de registros da consulta imprimindo os campos com as mesmas definições da coluna. Perceba que algumas colunas estão centralizadas e outras não. Na **Figura 3** vemos o relatório finalizado.

Conhecendo todas as possibilidades da classe FPDF

Bem, até agora viu-se os principais métodos da classe *FPDF* que, ao meu ver, são suficientes para criar a maioria dos relatórios. Mas ainda existe mais, veja a **Tabela 1** todos os métodos e a descrição das suas funcionalidades.

Conclusão

Nesse artigo apresentou-se a classe *FPDF* que possibilita, facilmente, criar



Método	Descrição
AcceptPageBreak	Orienta sobre aceitação ou não de quebra de página automática
AddFont	Possibilita a adição de nova fonte ao relatório
AddLink	Cria um link externo (http)
AddPage	Adiciona uma nova página ao relatório
AliasNbPages	Cria um apelido para variável que controla o número de páginas do relatório
Cell	Formata e imprime uma célula
Close	Fecha (termina) o relatório
Error	Cria um exceção do tipo fatal que aborta a execução do código restante
Footer	Método do rodapé que pode ser estendido na classe filha
FPDF	Nome da classe construtora (pai)
GetStringWidth	Retorna o tamanho do texto levando em conta a unidade de medida
GetX	Retorna a posição horizontal atual do cursor (relatório)
GetY	Retorna a posição vertical atual do cursor (relatório)
Header	Método do cabeçalho que pode ser estendido na classe filha
Image	Imprime uma imagem do tipo PNG, GIF e JPG
Line	Cria uma linha
Link	Insere um link criado pelo método *AddLink*
Ln	Força uma quebra de linha
MultiCell	Formata e imprime uma célula com várias linhas fazendo quebra automática
Output	Salva e/ou retorna o relatório. Em outras palavras apresenta o mesmo no browser
PageNo	Retorna o número da página
Rect	Cria um retângulo (desenho)
SetAuthor	Define o autor do documento
SetAutoPageBreak	Define o modo de quebra automática
SetCompression	Define se existe ou não compressão do relatório
SetCreator	Define o criador do documento
SetDisplayMode	Define o modo de visualização
SetDrawColor	Define a cor das linhas
SetFillColor	Define a cor de fundo (preenchimento)
SetFont	Define a fonte para escrita
SetFontSize	Define o tamanho da fonte
SetKeywords	Associa palavras chaves ao documento (utilizado para buscas)
SetLeftMargin	Define a margem esquerda do relatório
SetLineWidth	Define a largura da linha
SetLink	Define o destino de um link interno
SetMargins	Define as margens do relatório
SetRightMargin	Define a margem direita do relatório
SetSubject	Define o assunto do documento
SetTextColor	Define a cor do texto que será escrito
SetTitle	Define o título do documento
SetTopMargin	Define a margem superior do relatório
SetX	Seta a posição horizontal para a próxima escrita
SetXY	Seta a posição horizontal e vertical para a próxima escrita
SetY	Seta a posição vertical para a próxima escrita
Text	Imprime um texto (escreve)
Write	Imprime o texto na seqüência

Tabela 1. Métodos disponíveis na classe FPDF

relatórios no formato *PDF* para apresentação na *WEB*. Você pode aprender mais sobre essa classe vendo os tutoriais disponíveis no site do projeto: www.fpdf.org. Agora temos uma boa forma de criar relatórios no PHP sendo com *Delphi For PHP* ou não. Coloque a mão na massa e crie vários relatórios em seus sistemas ●

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

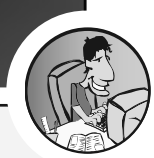
Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Nesta seção você encontra artigos sobre a linguagem PHP e a ferramenta Delphi for PHP

Orientação a Objetos no PHP

Conheça conceitos e novidades importantes



Resumo DevMan

Trabalhar com orientação a objetos tornou-se uma prática comum para vários desenvolvedores, por isso o presente artigo irá demonstrar algumas das características desta metodologia aplicada ao PHP, dicas e cuidados a respeito do funcionamento do modelo de objetos implementado no PHP e quando possível um paralelo com a linguagem Delphi. Além disso, descreverá as novidades referentes a objetos para a versão 5.3 que promete ser uma das mais proeminentes da linguagem.

Nesse artigo veremos

- Orientação a Objeto;
- Dicas sobre como trabalhar com objetos no PHP;
- Novidades para a versão 5.3 do PHP.

Qual a finalidade

- Apresentar os conceitos mais comuns sobre orientação a objetos aplicados ao PHP.

Quais situações utilizam esses recursos?

•Construir sistemas Web usando objetos ajuda o desenvolvedor a evitar replicação de código, já que uma classe ou várias classes podem ser aproveitadas em vários projetos.

Alexandre Altair de Melo

(alexandre@phpsc.com.br)

Bacharel em Sistemas de Informação pela UNIVILLE. Pós-Graduado em Gerenciamento e Planejamento Estratégico na PUCPR. Atua desde 2001 no desenvolvimento de sistemas Web e Desktop com as tecnologias (Delphi, Java e PHP). Possui as certificações: Java – SCJP 1.4 e PHP – ZCE 5.0. Atua como um dos coordenadores do grupo de usuários PHPSC e também como professor do Senac. É palestrante e instrutor de cursos na área Web. É co-autor do livro PHP Profissional – Aprenda a desenvolver sistemas profissionais orientados a objetos com padrões de projeto, editado pela Novatec.

Orientação a objetos é uma metodologia comum hoje e presente na maioria das linguagens de programação. Com o *PHP* não poderia ser diferente. A partir da versão 5, a linguagem apresentou grandes mudanças em sua estrutura interna para suportar esta técnica de forma mais consistente. Neste artigo apresentaremos alguns dos conceitos que foram adicionados nesta versão e as novidades para versão 5.3. Para aqueles que estão começando agora, é feita uma apresentação rápida dos três conceitos clássicos pertinentes a orientação a objetos, são eles: *herança, polimorfismo* e *encapsulamento*. O presente artigo não tem o objetivo de explorar ou esgotar todos os conceitos da orientação a objetos como classes, propriedades, etc, mas servir como uma revisão geral sobre este tema aplicado ao *PHP*.

Encapsulamento

O conceito de *encapsulamento* é uma característica da orientação a objetos que

permite que você oculte partes, independente da sua implementação. Assim, o *encapsulamento* permite que você construa partes ocultas do software que realizem uma funcionalidade e ocultem sua construção do mundo exterior. Isso se assemelha muito a uma caixa preta aonde você só tem acesso *interface*(*método*), porém não ao seu código.

Se compararmos ao Delphi for Win32, temos como exemplo de *encapsulamento* componentes, dlls, bpls e outros conjuntos de arquivos que podem ter parte de suas rotina ocultas.

Herança

Já o termo *herança* se assemelha ao mesmo conceito que existe no mundo real. A palavra se refere ao conceito de receber algo. Herança pode ser entendida como um mecanismo que permite construir uma nova classe com base em uma classe previamente existente, onde a nova classe herda automaticamente todos os atributos, comportamentos e implementações da sua classe *Pai*.

Polimorfismo

Polimorfismo significa muitas formas ou algo que pode mudar de forma. Em termos de programação, isso significa que podemos ter um mesmo método sendo executado de maneiras diferentes. Esse *polimorfismo* pode ser implementado em uma herança que permite que uma classe filha sobreponha o(s) método(s) de sua classe pai ou até mesmo dentro de uma mesma classe que apresente variações de um determinado método.

Em outras palavras, o polimorfismo pode ser usado para criar classes e objetos filho, que tenham algumas das funcionalidades e atributos modificados após a sua criação.

Definindo classes e criando objetos no PHP

Uma definição rápida para o conceito de classe é que ela se apresenta como um molde para criação de objetos definindo seus métodos e atributos. Já um objeto é uma instância desta classe, e que pode alterar os seus atributos e fazer o uso de seus métodos. Fazendo um exercício rápido para entender esse conceito pense na palavra *carro*, sua mente automaticamente projetou algo como tendo 4 *rodas, motor, portas*, etc. Essa projeção, que recebe o nome de *abstração,* definiu uma forma do que conhecemos como *carro,* e podemos chamar de *classe*. Agora imagine um estacionamento, ali teremos várias instâncias da classe *carro*, pois cada *carro* segue o padrão: 4 *rodas, motor, portas,* etc. Contudo cada *carro* tem características diferentes, porém todos pertencem a mesmas origem (classe). Isso é verdade porque nesse estacionamento podem existir carros com duas ou quatro portas, com as mais variadas cores, porém todos continuam sendo carros!

Trazendo isso para o contexto do *PHP,* para criar uma classe na linguagem basta fazer uso da palavra chave *class* e após a escrita deste comando deve-se escrever o nome da classe. Esse nome pode ser qualquer nome, contanto que não seja uma palavra reservada do PHP. Após o nome dela coloca-se um par de chaves *{}* que irá delimitar o corpo desta classe. Uma *pseudo* variável chamada *$this* fica disponível ao desenvolvedor quando este precisar chamar um método e/ou atributo dentro do contexto do objeto.

Para exemplificar melhor e entendermos o que foi dito, vamos criar um exemplo simples. Crie três scripts separados com os nomes *Imposto.php*, *ISS.php* e *resultados_classes.php*. Digite o código de cada um de acordo com as **Listagens 1, 2,** e **3** e salve-os dentro do mesmo diretório. Use o *Bloco de Notas* do *Windows* ou qualquer outro editor de sua preferência, como o *Dreamweaver* da *Adobe*, por exemplo.

Comecemos analisando a *pseudo* variável *$this* no *PHP,* que tem a mesma função de *self* no *Delphi*. Veja os códigos

Listagem 1. Criando a classe Imposto

```
<?php
class Imposto{
  private $aliquota;
 public function calcularImposto($preco, $aliquota) {
      $this->aliquota = $aliquota;
      return ($preco*$this->getAliquota())/100;
   }
 public function getAliquota() {
      return $this->aliquota;
   }
}
?>
```

Listagem 2. Criando a classe ISS que estende (Herda) de Imposto

```
<?php
require_once('Imposto.php');
class ISS extends Imposto{
  public function calcularImposto($preco, $aliquota){
      if (is_null($aliquota)){
           $aliquota = 5;
       }
      return parent::calcularImposto($preco, $aliquota);
   }
}
?>
```

Listagem 3. Criando objetos e um exemplo de polimorfismo na prática

```
<?php
require_once('Imposto.php');
require_once('ISS.php');
function imprimeInstancia(Imposto $imposto){
  if ($imposto instanceof ISS){
      echo "O objeto passado pertence a classe ISS<BR>";
 }elseif(
    $imposto instanceof Imposto){
      echo "O objeto passado pertence a classe Imposto<BR>";
 }else{
      echo "Objeto sem classe<BR>";
   }
}
$imposto = new ISS();
echo "ISS de um produto com valor de 15: " .
  $imposto->calcularImposto(15, null)."<BR>";
imprimeInstancia($imposto);
$imposto = new Imposto();
echo "Imposto de um produto com valor de 10 e taxa de 8.5%: " .
  $imposto->calcularImposto(10, 8.5)."<BR>";
imprimeInstancia($imposto);
?>
```

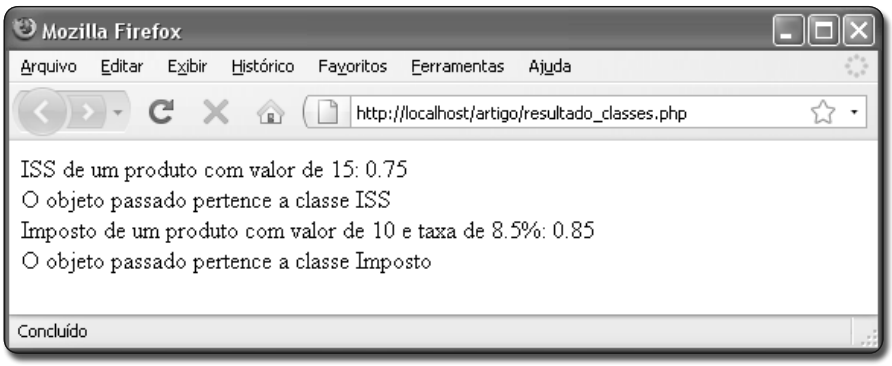


Figura 1. Resultado do código da Listagem 3

Listagem 4. Métodos construtores e destrutores em uma classe

```
<?php
class Pessoa{
  private $nome;
 private $idade;
 public function __construct($nome, $idade){
      $this->nome = $nome;
      $this->idade = $idade;
   }
 public function __destruct(){
      echo "Meu nome: ".$this->nome.". Tenho: ".$this->idade." anos";
   }
}
$pessoa = new Pessoa("Fulano", "21");
?>
```

das listagens mencionadas e o resultado na **Figura 1**. Perceba que criamos uma classe chamada *Imposto* na **Listagem 1.** A palavra reservada *$this* nas funções *calcularImposto* e *getAliquota*, fazem referência a variável *$aliquota* da classe *Imposto*. Já na **Listagem 2** estamos estendendo, ou seja, herdando da classe *Imposto* e criando uma nova classe, a *ISS*. Isso é um exemplo claro de *herança* como já explicado anteriormente. Usamos a função *require_once()* para importar o código da classe *Imposto* que foi salva no arquivo *Imposto.php*. Perceba o uso da palavra reservada *extends* para estender a classe *Imposto*.

```
class ISS extends Imposto{[…]
```

> **Nota**: O *PHP* também tem a palavra chave *self*, porém seu significado refere-se ao contexto de classe e não de um objeto em específico. Vale destacar também que há permissão para declarar mais de uma classe em um mesmo arquivo fonte.

A **Listagem 3** é ainda mais interessante, vejamos. Vemos um exemplo de criação de objetos. Para criar um novo objeto no PHP faz-se uso da palavra chave *new*. Repare que na função *imprimeInstancia* há uma indução de tipo, onde um objeto *Imposto* é esperado. Ali está um exemplo do *polimorfismo*, já que a função espera qualquer objeto, contanto que seja do tipo *Imposto*. Note que o método funciona tanto para a instância de *ISS* quanto para *Imposto*, já que *ISS* é um tipo de *Imposto*.

Execute o arquivo *resultados_classes.php* no seu *browser* preferido e veja que nosso script funciona perfeitamente. Em outras palavras: utilizando a palavra reservada *class*, criamos uma classe chamada *Imposto* e depois construímos o seu corpo dentro do escopo delimitado pelo par de chaves. Métodos em uma classe *PHP* podem ser codificados utilizando a notação:

```
visibilidade function <nome do_método>
  [(parâmetros)]
```

Vale destacar que, ao contrário do *Delphi*, não há a necessidade de especificar o tipo de retorno do método ou mesmo se ele não irá retornar nada. Você sabe se um método de uma classe irá retornar algo ou não, se este apresentar dentro do seu escopo a palavra chave *return*. Além disso, a assinatura de um método pode variar. Se o seu propósito for para representar métodos de classe adiciona-se ainda *static* ou se este delegar a sua implementação pode-se usar *abstract*. Sobre atributos a sua codificação é mais simples, geralmente coloca-se o operador de visibilidade acrescido do nome do atributo, também não é necessário indicar o tipo de dado que será armazenado. Com o uso da palavra chave *private*, que restringiu o acesso ao atributo *$aliquota*, somente para a própria classe aplicamos o conceito de *encapsulamento*. Há como especificar restrições de visibilidade em classes, e veremos mais adiantes nesse artigo.

> **Nota**: Apesar de não ser necessário indicar a visibilidade em um método ou atributo, seu uso é altamente recomendável.

> **Nota:** A palavra reservada *instanceof* é usada para determinar se uma variável do *PHP* pertence a determinada classe. Do lado esquerdo fica a variável que se quer avaliar e do lado direito a classe.

Construtores e Destrutores

Antes da versão 5 do PHP, o nome do construtor era o mesmo nome classe. A partir dessa versão, tem-se a introdução da palavra chave *__construct*. Vale ressaltar que o *PHP* só permite *um* construtor por classe. Caso não seja informado nenhum, a linguagem cria um construtor padrão. Para fazer chamadas ao construtor de uma classe *Pai*, por exemplo, é utilizada a palavra chave *parent*. Ex:

```
parent::__construct
```

O método *destrutor*, de um objeto será chamado assim que todas as referências a um objeto particular forem removidas da memória, quando o objeto for explicitamente destruído ou ainda em uma seqüência de destruição de objetos. Para definir um método *destrutor* em uma classe, deve-se usar a palavra chave *__destruct*. Assim como no construtor, para chamar o método *destrutor* de uma classe *Pai*, basta novamente

fazer uso de *parent* seguido do método *__destruct*. Experimente criar um novo *script* e digitar o código da **Listagem 4** que mostra um exemplo de uma classe com um *construtor* e *destrutor* definido. Execute-o e veja o resultado. Nesse *script*, estamos criando duas variáveis privadas, ou seja, de uso somente da classe *Pessoa* e fazendo referência a elas no método *__construct()* da classe. Em seguida, atribuímos a variável *$pessoa* a criação de um objeto *Pessoa*(*instância*). O método *__construct()* então associa a *nome* e *idade* o valor recebido em seus dois parâmetros, *$nome* e *$idade*, porém a impressão das informações são feitas somente no método *__destruct()*.

Se você executar o script verá que funciona normalmente, pois assim que a classe é criada a mesma é destruída (ao final da leitura do script pelo *browser*) mostrando então o *nome* e *idade* passados.

Nota: O *operador* **::** (dois pontos) é o operador que permite acesso a membros estáticos e constantes de classes.

Perceba que até aqui utilizamos a palavra reservada *public* em todas as classes. A visibilidade de um atributo ou método pode ser definida utilizando na sua declaração as palavras-chave: *public*, *protected* ou *private*. Itens declarados como *public* podem ser acessados por todo mundo. *Protected* limita o acesso a classes herdadas (e para a classe que define o item). *Private* limita a visibilidade para apenas a classe que define o item (*método*/*atributo*). O *PHP* não tem o conceito de *published* do *Delphi*.

Classes abstratas e interfaces

Interfaces de objetos são classes especiais que podem ser definidas no código. Elas especificam um ou mais métodos e seus respectivos parâmetros, mas ao contrário de classes tradicionais, não os codificam. Toda interface é definida com a palavra reservada *interface*, seguida do seu nome que possui as mesmas regras de definição do nome de uma classe e sua respectiva lista de métodos e parâmetros. Uma classe que deseja implementar uma interface deve usar a palavra chave *implements* na sua declaração.

Figura 2. Resultado do código da Listagem 6

Listagem 5. Classe abstrata simulando operações de uma conta

```
<?php
abstract class Conta{
  protected $saldo;
  public function __construct($saldo){
      $this->saldo = $saldo;
  }
public abstract function sacar($valor);
public abstract function depositar($valor);
public abstract function mostraSaldo();
}
?>
```

Listagem 6. Classe ContaPoupanca que estende de Conta e implementa ImpostoConta

```
<?php
require_once('Conta.php');
require_once('ImpostoConta.php');
class ContaPoupanca extends Conta implements ImpostoConta {
  public function __construct($saldo) {
      parent::__construct($saldo);
  }
 public function sacar($valor){
      $this->saldo = $this->saldo - ($valor + $this->
       calcularImposto($valor, ImpostoConta::CPMF));
  }
 public function depositar($valor) {
      $this->saldo = $this->saldo + $valor;
  }
 public function mostraSaldo() {
      echo "Saldo da conta: ".$this->saldo."<br>";
  }
 public function calcularImposto($valor, $taxa) {
      return round((($valor*$taxa)/100),2);
  }
}
$poupanca = new ContaPoupanca(100);
$poupanca->sacar(10);
$poupanca->mostraSaldo();
$poupanca->depositar(10.04);
$poupanca->mostraSaldo();
?>
```

Nota: O *PHP* não tem herança múltipla. Para simular este conceito, a linguagem faz uso de interfaces. Uma interface tem todas as assinaturas de métodos como públicos mesmo que o desenvolvedor não especifique isso.

Já o conceito de classes abstratas foi introduzido na versão 5. A explicação de seu funcionamento é muito parecida com o de uma interface, contudo esse tipo de classe tem algumas características particulares como:

- Não é permitido instanciar classes abstratas, se isto for feito irá causar um erro;
- Qualquer classe que contiver pelo menos um método que seja abstrato deverá ser declarada abstrata. Para isso o uso da palavra reservada *abstract* se faz necessário tanto na declaração do nome da classe como na assinatura do método;
- Classes abstratas podem conter tanto métodos abstratos que contenham somente a assinatura do método, como métodos concretos;
- Qualquer classe que estender de uma classe abstrata deve implementar os métodos abstratos contidos nesta;
- Quando uma classe abstrata implementa uma interface, ela não precisa

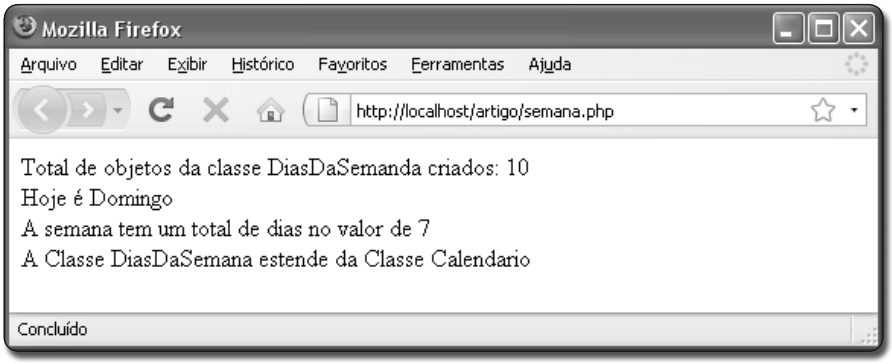


Figura 3. Resultado do código da Listagem 8

Listagem 7. Uso da palavra chave self e static além de constantes de classe

```
<?php
class Calendario {
  const NOME = "Classe Calendario";
}
class DiasDaSemana extends Calendario {
 const DOMINGO = "Domingo";
 const SEGUNDA_FEIRA = "Segunda-Feira";
 const TERCA_FEIRA = "Terça-Feira";
 const QUARTA_FEIRA = "Quarta-Feira";
 const QUINTA_FEIRA = "Quinta-Feira";
 const SEXTA_FEIRA = "Sexta-Feira";
 const SABADO = "Sábado";
 const NOME = "Classe DiasDaSemana";
 const TOTAL_DE_DIAS = 7;
 static $instancias = 0;
 private function __construct() {
      self::$instancias++;
   }
 public static function geraInstancia() {
    new self();
   }
 public static function imprimeConstantesNome() {
   return "A ".self::NOME." estende da ".parent::NOME;
   }
}
for ($i = 0; $i < 10; $i++) {
  DiasDaSemana::geraInstancia();
}
echo "Total de objetos da classe DiasDaSemanda criados: ".DiasDaSemana::$instancias."\n";
echo "Hoje é ".DiasDaSemana::DOMINGO."\n";
echo "A semana tem um total de dias no valor de " . DiasDaSemana::TOTAL_DE_DIAS."\n";
echo DiasDaSemana::imprimeConstantesNome();
?>
```

codificar seus métodos deixando isso a classe concreta que estender dela, conforme já comentando no tópico de interfaces.

Veja um exemplo do uso de interfaces no código a seguir e nas **Listagens 5** e **6** . O resultado, podemos ver na **Figura 2**.

```
<?php
interface ImpostoConta{
  const CPMF = 0.38;
  public function calcularImposto($valor,
$taxa);
}
?>
```

Aqui definimos uma interface que define um método para calcular imposto, toda a classe que implemente uma interface deve implementar os métodos constantes nesta. A exceção se dá se a classe for declarada abstrata assim a implementação fica a cargo da primeira classe concreta da hierarquia de classes.

Definimos um classe base denominada *Conta* e que possui uma modelagem das operações normalmente encontradas numa conta de banco que são *sacar, depositar* e *mostrar o saldo.*

Por último criamos a classe *ContaPoupanca* e implementamos todos os métodos tanto da interface como da classe pai. Usamos a função *round* para arredondar o cálculo do imposto em duas casas no método para *calcularImposto.* O leitor pode reparar também que no código da classe *ContaPoupanca* os métodos herdados da classe *Conta* tiveram a palavra reservada *abstract* retiradas de sua assinatura, já que a classe iria implementar os mesmos.

Outros conceitos úteis

Vamos continuar listando mais algumas novidades da versão 5 referente a objetos. Uma delas foi a introdução de constantes de classe e sua declaração, que é feita através da palavra chave *const*. Para acessar constantes de classe fora do escopo dela a seguinte notação é utilizada:

```
NomeClasse::Constante
```

E já dentro da classe via a notação:

```
self::Constante
```

A palavra reservada *self* também pode ser utilizada para se referenciar a própria classe. Veja o exemplo da **Listagem 7,** onde é feito o uso de constantes de classe e *self,* o resultado é visto na **Figura 3**. Para codificar constantes em uma classe utilize *const*.

No exemplo da **Listagem 9**, criamos primeiramente uma classe *Calendario* e definimos uma constante dentro dela. Já na classe *DiasDaSemana* especificamos que esta devia estender de *Calendario,* além de criar várias constantes para armazenar o nome dos dias da semana e total de dias. Usamos um atributo estático nesta classe para guardar a quantidade de objetos criados. Também definimos o construtor como privado. Isso é usado para simular o padrão de projeto *Singleton*, e fizemos a criação de vários objetos via o método *geraInstancia*. Com isso exemplificamos mais algumas funcionalidades de orientação a objetos disponíveis no *PHP*.

Vamos comentar um pouco sobre algumas particularidades ao se trabalhar com objetos no PHP. Um ponto que deve

Nota do DevMan

Os padrões de projeto, ou *Design Patterns*, são soluções para problemas recorrentes que surgem durante o desenvolvimento de um projeto. O padrão citado, *Singleton*, define uma solução para situações onde se deseja garantir que apenas uma instância de um determinado objeto exista durante toda a execução da aplicação, além de especificar um único local de acesso a esse objeto.

ser explicado refere-se à questão de quanto tempo um objeto PHP dura em memória. O tempo de vida desse objeto dura enquanto durar a execução do *script* PHP, já que não temos uma *memória* no estilo de ambientes de desenvolvimento como o desktop. Para que seja possível aproveitar um objeto já criado, devemos armazená-lo, por exemplo, em uma sessão. Porém também devemos tomar cuidado ao recuperar este objeto da sessão, já que a declaração da sua classe deve constar no ponto onde este é recuperado. Caso essa declaração não esteja presente, o objeto recuperado será do tipo *stdClass* do PHP, não tendo assim nenhuma das suas funcionalidades especificadas na classe original disponível. Para fazer essa declaração basta incluir a chamada do arquivo que contêm a definição da classe via funções *include_once* ou *require_once*, por exemplo. A função *require_once* foi utiliza no exemplo da **Listagem 2**.

Novidades para a versão 5.3

Após essa visão geral sobre objetos vamos falar um pouco sobre o futuro. O *PHP 6* ainda vai demorar um pouco para ser lançado, contudo conforme o que se comenta nas listas da comunidade essa versão deve sair durante o ano de 2009. A boa notícia é que a versão 5.3 que esta para sair no último trimestre de 2008, contém inúmeras novidades esperadas pelos desenvolvedores há muito tempo. Dentre elas existem duas referentes a objetos. A questão de *Namespaces* e *Garbage Collector* para o gerenciamento de memória. Vamos a elas:

• *Namespaces*: Essa era uma das funcionalidades mais pedidas e também uma das que causava mais controvérsias. Originalmente prevista para sair no PHP6, foi adiantada para a versão 5.3. Assim, essa versão contará com a implementação completa de *namespaces*, com suporte a *autoload* e *namespaces* hierárquicos. Isso permite aos desenvolvedores organizar melhor o código, evitando conflitos com bibliotecas internas do PHP ou de terceiros, criação de nomes de classes mais simples e principalmente produzindo um código mais limpo e legível. Essa funcionalidade ajudará muito o desenvolvimento de *frameworks*.

Em suma, com a introdução de *namespaces* no PHP você poderá fazer um código como mostrado a seguir:

```
namespace semana;
class Dia{
   const INICIO_SEMANA = 'Domingo';
   public function executar(){
     echo self::INICIO_SEMANA;
   }
}
```

E seu uso poderá ser feito como segue. Veja que fazemos referência ao namespace semana diretamente na chamada da função *executar()*. Obviamente, o *namespace* semana poderá estar declarado no mesmo script PHP onde é chamado ou ainda em outro arquivo, requerendo claro sua inclusão no *script* usando *include(), include_once(), require()* ou *require_once()*.

```
$objeto = semana::Dia::executar();
```

Há ainda a possibilidade de se usar o *namespace semana* usando a palavra reservada *use*, semelhante ao Delphi. Veja:

```
use semana;
$var = new Dia::executar();
```

• *Garbage Collector*: Agora é possível fazer limpeza da memória que não é mais utilizada de uma forma mais direta, reduzindo os problemas como o fatídico *Memory allocation error*. As funções *gc_enable* e *gc_disable* vieram com essa finalidade.

Listamos duas das novidades referentes a objetos, porém esta versão contará com muitas outras como a melhora de performance em torno de 15%, uma nova biblioteca para conexão com *MySQL* entre outras novidades.

Para mais informações visite *wiki.php.net/todo/php53*, ali está listado o planejamento da versão e o que já foi feito para esse lançamento. Estas funcionalidades, que serão disponibilizadas, mostram que o *PHP* está em constante evolução, sempre atento às necessidades da comunidade e do mercado.

Conclusão

Durante o artigo focamos em apresentar alguns dos conceitos sobre orientação a objetos existentes no PHP. Existe muito mais além do que foi descrito no artigo como clonagem de objetos, reflexão, métodos mágicos, padrões de projetos entre outros assuntos interessantes. E de forma alguma como já dito no início gostaríamos de esgotar o assunto. O estudo não termina por aqui, uma visita ao site da linguagem vale a pena, visite *www.php.net*.

O manual on-line constante no site descreve vários exemplos e é uma ótima fonte de consulta para saber o que é possível fazer com objetos e *PHP*. Adicione também a leitura de livros, revistas, fóruns na internet que só tem a acrescentar ao processo de aprendizagem. O artigo também mostrou que o *PHP* está evoluindo sempre tentando agregar em seu núcleo boas práticas já implementadas em outras plataformas.

Finalizando pode-se dizer que OO é uma ótima prática para construção de sistemas. E que independente da linguagem uma vez aprendido o seu conceito, o desenvolvedor tem material para trabalhar com qualquer tecnologia, bastando, entretanto aprender a gramática de cada ferramenta. Espero ter contribuído no seu processo de conhecimento para entender OO + *PHP*.

Ao leitor saudações, bons estudos e até a próxima! ●

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Nesta seção você encontra artigos para iniciantes na linguagem Delphi

Criando service application

Crie aplicações e faça as rodar como Serviços no Windows

Maikel Marcelo Scheid
(maikelscheid@gmail.com)
é técnico em Informática com ênfase em Análise e Programação de Sistemas. Atua na área de Desenvolvimento de Softwares em Delphi para plataforma Win32 e .NET com banco de dados Firebird e MS SQL.

Quando desenvolvemos aplicações *Win32* ou *.net*, estas são executadas sempre a partir de solicitações voluntárias de usuários que fazem as chamadas ao sistema. Mas há situações em que necessitamos criar aplicações capazes de funcionarem sem a interação do usuário e de forma independente, ou seja, que sejam executadas de forma automática, como os serviços do *Windows*. As *Service Applications* como também podem ser chamadas, iniciam juntamente com o sistema operacional, independente de haver ou não algum usuário *logado* no computador ou mesmo interagindo com o sistema. Ao acessarmos o ícone *Serviços* dentro de *Ferramentas Administrativas* encontrado no *Painel de Controle* do *Windows* podemos ver que diversos serviços são listados, uns em execução e outros não. Todos esses são considerados *Service Applications*. Perceba que temos serviços de tudo que é jeito, como por exemplo, *Horário do Windows*.

Resumo DevMan

A criação de serviços no Windows é um recurso bastante útil na execução de processos de forma automática sem intervenção de usuários. Podemos citar como exemplos de serviços os itens Horário do Windows, Log de eventos do Windows, e uma série de outros.
Nesse artigo veremos a criação de um serviço de backup automatizado com opção de compactação usando WinRAR e Winzip.

Nesse artigo veremos

- Criação de um serviço de backup de arquivos;
- Cópia e compactação dos arquivos para backup;
- Criação e instalação do serviço;
- Gerenciamento do serviço no painel de controle.

Qual a finalidade

- Entender os processos e implementação de serviços no Windows.

Quais situações utilizam esses recursos?

- Em situações onde se faz necessário que o aplicativo esteja sempre em execução, não dependendo da ação de usuários para iniciar o processo.

Neste artigo veremos criaremos um pequeno utilitário para backup de arquivos comuns. Primeiramente, faremos a criação de uma aplicação simples *Win32* e em seguida faremos a conversão transformando-a em uma *Service Application*.

Desenvolvendo o aplicativo

Como mencionado criaremos uma aplicação *Win32* para backup manual e compactação dos arquivos. Para isso usaremos o *Delphi 7,* porém nada impede que seja utilizado outro *Delphi* a sua escolha. Vamos iniciar criando a aplicação usando o menu *File>New>Application* e salvando sua *Unit* principal como *uBackup.pas* e o projeto como *prjBackup.dpr* em um diretório do seu computador. Altere a propriedade *Name* do formulário para *frmServiceBackup* e o *Caption* para *Configuração de Serviço de Backup*. Para melhor entender e organizar nossa aplicação de backup, utilizaremos quatro componentes *GroupBox* da paleta *Standard* que deverão ser arrastados para o formulário.

No primeiro *GroupBox (gbBackup),* altere a propriedade *Caption* para *Backup* e arraste para dentro deste um *Label* com o *Caption* de *Localizar arquivos de Origem* para identificar a busca dos arquivos. Ao lado deste, adicione um *Button (btnSelFiles)* que fará a chamada de um diálogo de pesquisa dos arquivos. Adicione também dentro do *gbBackup* um componente *ListBox (lbListaBackup)* no qual iremos exibir os arquivos da lista. Por último, inclua mais dois componentes *Button (btnRemover* e *btnBkpManual)* alterando a propriedade *Caption* dos mesmos seguindo a ordem para *Remover da Lista* e para *Backup Manual*. Arraste ainda ao formulário um componente *OpenDialog(dlgAddFiles)* da paleta *Dialogs* que será utilizado para exibir o diálogo de busca para inclusão dos arquivos na lista, alterando sua propriedade *Options>ofAllowMultiSelect* para *True* possibilitando que vários arquivos sejam selecionados de uma única vez.

Proporcionaremos nesta aplicação uma configuração onde o usuário poderá optar em apenas copiar os arquivos da lista para um novo diretório ou a compactação dos mesmos utilizando os compactadores *WinRAR* ou *WinZip* armazenando os arquivos compactados também no diretório a ser definido. Para esta opção adicione um novo *GroupBox(gbConf)* alterando também sua propriedade *Caption* para *Configurações* e arraste para dentro do mesmo um componente *CheckBox(ckbUsarCompact)* que deverá alterar o *Caption* para *Usar Compactação dos Arquivos?*.

Para a configuração de escolha dos tipos de compactação, adicione imediatamente abaixo ao *gbConf* um componente *RadioGroup(rgTipoCampact)* alterando sua propriedade *Caption* para *Tipo de Compactação* e a propriedade *Items* onde deverá incluir as duas opções disponíveis, sendo uma em cada linha da caixa de texto que será exibida: *WinRAR* e *WinZip*. Altere ainda o *ItemIndex* para zero fazendo com que o primeiro item permaneça selecionado e defina *Visible* para *False*.

No último *GroupBox(gbDestino)* a ser adicionado, deverás alterar a propriedade *Caption* para *Destino do Backup* e para dentro do mesmo arraste um componente *Edit(edtDestino)* que irá armazenar o diretório destino onde será realizado o backup da aplicação, podendo ser fixado na propriedade *Text* informando um caminho de diretório, como por exemplo *C:\Backup*.

Pelo fato de estarmos criando um aplicativo com a idéia de deixá-lo automatizado e mais tarde transformá-lo em um serviço, precisamos utilizar alguma forma onde o usuário faça a configuração do sistema uma única vez e estas permaneçam gravadas para que no momento de *start* do serviço, ele saiba todas as rotinas a serem executadas sem que haja intervenção do usuário. Para isto utilizaremos arquivos de configuração, ou arquivos *INI* como são comumente chamados.

Mais precisamente, um arquivo *INI* é um arquivo de texto onde definimos regras que serão carregadas e reescritas pelo sistema, regras estas que serão utilizadas para a configuração automática do sistema. Faremos a configuração destas regras em um *Button(btSalvarConfs)* que deverá ser adicionado ao final do formulário com sua propriedade *Caption* alterada para *Salvar Configurações*, que irá concluir o processo da criação do visual do aplicativo de backup (**Figura 1**).

Para criar um arquivo *INI* iremos utilizar o bloco de notas (*notepad*) que deverá ser aberto e digitar a seguinte estrutura:

```
[BACKUP]
COMPACTAR=S
TIPO=Z
DESTINO=C:\Backup\tmp
ULTIMO=28/08/2008
```

Nota do DevMan

Você sabia que o *Windows* possui uma série de serviços automatizados para realizar as mais diversas tarefas do dia-a-dia? Pois bem, algumas das tarefas mais importantes para o bom funcionamento do sistema operacional são na verdade serviços que ficam executando em *BackGround*, ou seja, ocultos sem que o usuário perceba. Um exemplo bastante claro disso é a detecção automática de atualizações, que encontramos no painel de *Serviços* com o nome *Windows Update*. Esse serviço é responsável por verificar qual a versão atual de nosso Windows bem como seus componentes e compará-los com as versões presentes no servidor da Microsoft. Quando uma atualização é detectada o serviço avisa o usuário que pode optar ou não pela instalação do Update.

ClubeDelphi PLUS www.devmedia.com.br/clubedelphi/portal.asp

Acesse agora o mesmo o portal do assinante ClubeDelphi e assista a uma vídeo aula de Jefferson Junglaus que mostra como trabalhar com serviços no Windows.

http://www.devmedia.com.br/articles/viewcomp.asp?comp=3381

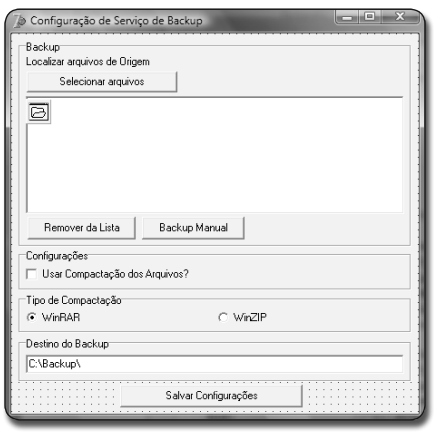


Figura 1. Visual da aplicação de backup

Listagem 1. Adicionando arquivos selecionados à lista.

```
procedure TfrmServiceBackup.btnSelFilesClick(Sender: TObject);
var
  i : integer;
begin
  if dlgAddFiles.FileName <> '' then
    for i := 0 n dlgAddFiles.Files.Count -1 do
      lbListaBackup.Items.Add(dlgAddFiles.Files[i]);
end;
```

Listagem 2. Salvando as configurações do sistema

```
procedure TfrmServiceBackup.btSalvarConfsClick(
  Sender: TObject);
var
  Confs : TIniFile;
begin
  Confs := nil;
  try
    Confs := TIniFile.Create(ExtractFilePath(Application.ExeName)
      + 'backup.ini');
    if ckbUsarCompact.Checked then
    begin
      Confs.WriteString('BACKUP','COMPACTAR','S');
      if rgTipoCampact.ItemIndex = 0 then
        Confs.WriteString('BACKUP','TIPO','R')
      else
        Confs.WriteString('BACKUP','TIPO','Z');
    end
    else
      Confs.WriteString('BACKUP','COMPACTAR','N');
    Confs.WriteString('BACKUP','DESTINO',edtDestino.Text);
  finally
    Confs.Free;
  end;
lbListaBackup.Items.SaveToFile(ExtractFilePath(Application.ExeName)
  +'lista.ini');
end;
```

Nesta estrutura estamos criando um novo bloco chamado de *Backup* para armazenar as informações de configuração do sistema. Neste bloco criamos itens, onde o primeiro item identificado como *Compactar* diz respeito à compactação dos arquivos, variando de acordo com as opções do usuário:

S: habilitar compactação;
N: desabilitar a mesma.

Para o item *Tipo*, caso tenha optado pela compactação dos arquivos, irá definir se o compactador a ser utilizado é o *WinZIP* (armazenado como *W*) ou o *WinRAR* (armazenado como *R*). No item *Destino* ficará por sua vez armazenado o diretório de destino dos arquivos e a opção *Ultimo* será a responsável por armazenar a data do último backup, evitando assim que o mesmo seja repetido em um único dia. Salve agora o arquivo do bloco de notas como *backup.ini* no mesmo diretório onde se encontram os arquivos fontes da aplicação onde posteriormente será localizado.

Bom, até o momento entendemos como o sistema cliente (assim o iremos chamar) irá funcionar, e iniciaremos agora o processo de codificação do aplicativo. Nesta codificação direcionaremos nessa primeira etapa a construção de processos simples que formarão uma lista de arquivos, opções e formas de compactação. Criaremos um processo manual para cópia dos arquivos, que depois iremos implementar no aplicativo de serviço, que fará a leitura destas configurações e acionará um processo automático.

A primeira codificação que iremos desenvolver será o processo de montagem dos arquivos na lista de backup, ou seja, utilizaremos o componente *dlgAddFiles* para buscar os arquivos e adicioná-los a lista adicionando o código da **Listagem 1** ao evento *OnClick* do *btnSelFiles*. Logo após a execução do componente de diálogo, percorremos por meio de um comando *for* todos os arquivos selecionados adicionando-os um a um ao componente *ListBox* formando a lista de arquivos do backup.

No próximo passo do processo de codificação, veremos como excluir arquivos adicionados indesejadamente a lista de backup, lembrando que apenas temos uma lista em memória formada. Isso significa que podemos excluir pelo próprio componente *ListBox* apenas adicionando o código a seguir ao evento *OnClick* do *btnRemover*, que irá executar o comando *Delete* no índice selecionado do componente.

```
lbListaBackup.Items.Delete(
  lbListaBackup.ItemIndex);
```

Nesse ponto já estamos criando e editando a lista dos arquivos para o backup e precisamos codificar as opções disponibilizadas nele, ou seja, quanto ao uso de compactação. No inicio do artigo, enquanto criávamos o layout do aplicativo definimos como *False* a propriedade *Visible* das opções de compactação (*WinZIP* ou *WinRAR*), mas que deverão ser exibidas ao selecionarmos a opção de compactação, para a qual se faz necessário adicionar o código seguinte no evento *OnClick* do componente *ckbUsarCompact* que irá habilitar ou desabilitar a visibilidade do componente *rgTipoCampact* de acordo com sua propriedade *Checked*.

```
rgTipoCampact.Visible :=
  ckbUsarCompact.Checked;
```

Como próximo passo na continuação de criação da nossa aplicação, codificaremos a rotina responsável por salvar as configurações do sistema no arquivo *Backup.ini* criado anteriormente. Nosso primeiro passo será declarar nas *Uses* do Delphi a *Unit IniFiles*, responsável pelo acesso e leitura dos arquivos de configurações. Realizada a declaração, adicione ao evento *OnClick* do botão *btSalvarConfs* o código da **Listagem 2** iniciando pela declaração de uma variável. É interessante que todas as variáveis de objetos criadas durante a programação sejam inicializadas com o *nil*, que faz com que as mesmas sejam criadas sem nenhum tipo de valor, assim não provocando *Warnings* (avisos) durante a compilação do aplicativo.

Na criação dos objetos recuperamos o caminho do executável do aplicativo, lugar onde obviamente encontraremos o arquivos **.INI*. Criado e conectado ao arquivo, utilizaremos a função *WriteString* para reescrever os valores no arquivo. Realizadas estas operações de gravação, limpamos o objeto criado na memória e

em seguida iniciamos o salvamento da lista de arquivos do backup, estes de forma bem mais simples a serem tratados por já estarem distribuídos em linhas e bastando apenas aplicar o *SaveToFile* do próprio *ListBox* para salvar todas as linhas em um novo arquivo de configurações **.INI*.

Salvando todas as configurações em arquivos **.INI* faz-se também necessário que estas mesmas configurações sejam carregadas quando o sistema for aberto novamente, ou seja, todas as opções que você salvou precisam voltar no mesmo estado na próxima vez que abrir o sistema. Para este procedimento, codificaremos o evento *OnCreate* do formulário (**Listagem 3**), para o qual definiremos um objeto do tipo *TIniFile* que será criado em *runtime* e fará a leitura do arquivo *backup.ini* que contém as configurações do sistema. Na medida em que os blocos de registros são lidos, combinamos as informações e realizamos a configuração do sistema, e logo em seguida a liberação do objeto criado da memória. No último segmento do código, utilizamos o *LoadFromFile* do componente *ListBox* e carregamos a lista de arquivos para backup, escrevendo-os da mesma forma como foram salvos dentro do componente.

Só para entendermos bem o código-fonte, estamos instanciando/criando o arquivo *.INI* em memória através do método *Create* da classe *TIniFiles*.Criaremos agora uma função que fará um backup manual de todos os arquivos da lista, a mesma função que utilizaremos logo mais para criação do serviço automatizado. Esta função será baseada nas atuais opções selecionadas no sistema, e de acordo com as mesmas tratará a lista de backup.

Acessando o código da aplicação, declare na seção *private* da *Uses* a *procedure Backup* a seguir que receberá como parâmetro o caminho dos arquivos a serem adicionados no backup.

```
procedure Backup(CaminhoArquivo: string);
```

Declarada a função na seção *private* da *uses* utilize o atalho *Shift+Ctrl+C* para que o Delphi no crie o cabeçalho da *procedure*. Adicione o código da **Listagem 4**. O código fará uma primeira verificação quanto ao campo de compactação, se a mesma será necessária ou apenas a cópia de arquivos será o suficiente. Para o caso da opção de compactação não estar habilitada, a rotina apenas fará a cópia dos arquivos para o diretório de destino do backup. Caso a opção de compactação esteja como *Checked = True* o sistema fará uma nova verificação, a fim de saber se o tipo de compactação será com o *WinZIP* ou *WinRAR*. Para qualquer uma das duas situações de compactação o processo será muito semelhante, montamos uma lista temporária, editamos o nome do arquivo e atribuímos a ele a extensão correspondente e criamos uma linha de comando para fazer referência aos diretórios de instalação.

Com o destino do backup, nome do arquivo e linha de comando invocamos a função *WinExec* da *API* do *Windows*. Esse comando é capaz de executar programas externos. Para melhor facilitar o entendimento da função, o código encontra-se comentado de acordo com as funcionalidades executadas em cada linha do comando.

A princípio criamos algumas variáveis para diversas tarefas ao longo do

Listagem 3. OnCreate do formulário

```
procedure TfrmServiceBackup.FormCreate(Sender: TObject);
var
  Confs : TIniFile;
begin
  Confs := nil;
  try
    Confs := TIniFile.Create(ExtractFilePath(Application.ExeName)
      +'backup.ini');
    if Confs.ReadString('BACKUP','COMPACTAR','') = 'S' then
    begin
      ckbUsarCompact.Checked := True;
      if Confs.ReadString('BACKUP','TIPO','') = 'R' then
        rgTipoCampact.ItemIndex := 0
      else
        rgTipoCampact.ItemIndex := 1;
    end
    else
      ckbUsarCompact.Checked := False;
    edtDestino.Text := Confs.ReadString('BACKUP','DESTINO','');
  finally
    Confs.Free;
  end;
  lbListaBackup.Items.LoadFromFile(ExtractFilePath(Application.ExeName)
    +'lista.ini');
end;
```

Listagem 4. Procedure que fará o backup

```
procedure TfrmServiceBackup.Backup(CaminhoArquivo: string);
var
  NomeArquivo: string;
  Temp: string;
  LinhadeComando: string;
  Destino: string;
begin
  if ckbUsarCompact.Checked then
  begin
    if rgTipoCampact.ItemIndex = 0 then
    begin
      Temp := ChangeFileExt(CaminhoArquivo, '.rar');
      NomeArquivo := PASTA_DESTINO_BKP + ExtractFileName(temp);
      LinhadeComando := 'C:\Program Files (x86)\WinRAR\WINRAR.EXE a "' +
        NomeArquivo + '" ' + CaminhoArquivo ;
    End
    else
    begin
      Temp := ChangeFileExt(CaminhoArquivo, '.zip');
      NomeArquivo := PASTA_DESTINO_BKP + ExtractFileName(temp);
      LinhadeComando := 'C:\Program Files (x86)\WinZip\WINZIP32.EXE -A "' +
        NomeArquivo + '" ' + CaminhoArquivo;
    end;
    try
      WinExec(Pchar(LinhadeComando),1);
    except
    end;
  end
  else
  begin
    Destino := PASTA_DESTINO_BKP + ExtractFileName(CaminhoArquivo);
    if not CopyFile(PChar(CaminhoArquivo), PChar(Destino), true) then
      ShowMessage('Erro ao copiar ' + CaminhoArquivo + ' para ' + Destino);
  end;
end;
```

Listagem 5. Executando um backup manual

```
procedure TfrmServiceBackup.btnBkpManualClick(Sender: TObject);
var
  I: Integer;
  CaminhoArquivo: string;
begin
  if ckbUsarCompact.Checked then
  begin
    for I := 0 to lbListaBackup.Count - 1 do
      CaminhoArquivo := '"'+lbListaBackup.Items.Strings[I] + '" ' +
        CaminhoArquivo;
    Backup(CaminhoArquivo);
  end
  else
  begin
    for I := 0 to lbListaBackup.Count - 1 do
    begin
      CaminhoArquivo := lbListaBackup.Items.Strings[I];
      Backup(CaminhoArquivo);
    end;
  end;
end;
```

Figura 2. Criando Service Application

procedimento. Em seguida verificamos o estado do *CheckBox* referente a compactação. Caso esteja checado, o sistema fará a verificação de qual compactador está selecionado e então montará uma *LinhaDeComando* para ser executada mais adiante. Por fim o procedimento chama o método *WinExec* passando o comando montado para ser executado.

Criada a *procedure* veremos como chamar ela a partir do backup manual. Digite o código da **Listagem 5**, no qual evento *OnClick* do botão *btnBkpManual*, no qual realizamos novamente uma verificação quanto a compactação dos arquivos, onde se habilitada fará um laço entre todos os arquivos da lista de backup adicionando-os um ao lado do outro e separados por aspas duplas, a fim de que a compactação seja realizada em um único arquivo.

Perceba que o processo é bastante simples. Apenas testamos novamente o *CheckBox* de compactação e percorremos o *ListBox* montando uma única linha de comando. Finalizada esta etapa da codificação, execute o sistema e teste quanto ao funcionamento de salvar e abrir o sistema com as configurações e da mesma forma a adição de arquivos na lista e o backup manual dos mesmos. Concluídos os testes, passaremos enfim para a criação do serviço, que verá ser um aplicativo simples com poucas codificações.

Criando o serviço

Agora faremos a parte mais importante do nosso sistema. Faremos sua conversão para *Service Application*. No *Delphi 7*, e outras versões, um tipo específico de projeto que pode ser criado. Acessamos essa opção através do menu *File>New>Other>Delphi Projects>Service Application* (**Figura 2**). Serão criados automaticamente pelo Delphi duas *Units*, uma referente ao *Source Code* e outra ao *Project*. Salve os respectivos documentos como *uServico.pas* e *prjService.dpr* no mesmo diretório onde criamos o aplicativo cliente do backup, pois faremos uma referência em *runtime* para localizar, ler e executar os arquivos já criados.

Nos documentos criados pelo *Delphi*, selecione também o serviço propriamente dito e altere sua propriedade *Name* para *mServico*. Podemos observar um diferencial no código do projeto, onde já nas declarações das *Uses* não temos mais presente a *Unit Forms* encontrada em todas as aplicações normais *Win32*. No lugar notamos a *Unit SvcMgr* específica para inicialização de serviços, ou seja, a execução do aplicativos acontece antes da autenticação de usuários, além disso, o único objeto que deve ser criado na inicialização é o serviço. Veja um trecho de código no *Source* do projeto:

```
if not Application.DelayInitialize or
Application.Installing then
    Application.Initialize;
Application.CreateForm(TmServico, mServico);
Application.Run;
```

O serviço na verdade não possui *form* e sim uma espécie de *Data Module* onde os componentes visuais são colocados e codificados. Selecionado o *mServico* (nosso Data Module) faremos a configuração das propriedades do serviço propriamente dito. Essas propriedades serão levadas em conta no momento da execução do serviço pelo sistema operacional, tais como: *nome de identificação do serviço* na lista de serviços do *Windows*, *dependências*, *permissões*, entre outros.

Iniciando pela primeira propriedade disponível, temos duas opções de permissão *AllowPause* e *AllowStop* permitindo que por meio de intervenções de usuários ou outros aplicativos o serviço possa ser pausado ou até mesmo parado, respectivamente. Em seguida temos em *Dependencies* a disponibilidade de adicionar uma lista de outros serviços, ou seja, pré-requisitos para que o serviço seja inicializado, ponto este que apenas acontecerá quando todas suas dependências definidas já estiverem rodando no sistema.

Para adicionar dependências, abra o editor da propriedade e adicione uma nova dependência (*Add New(Ins)*), selecione a dependência adicionada e na sua propriedade *Name* marque qual o serviço deverá ser o pré-requisito para execução do sistema, *FirebirdServerDefaultInstance* para definir o *Firebird* como pré-requisito (**Figura 3**).

Na propriedade *DisplayName* iremos definir uma descrição que identificará o serviço entre os demais da lista de serviço do *Windows*, defino-o como *Serviço de Backup ClubeDelphi*. Nas demais propriedades poderíamos ainda definir se o serviço será ou não interativo com o usuário, definir ainda o tipo do servi-

ço, modo de inicialização, entre outras propriedade que poderão ser mantidas como *default* para este exemplo.

Sendo um sistema baseado em tempo, ou seja, um backup por dia faremos o controle das verificações através de um *Timer* (*TimerBackup*) que deverá ser adicionado ao *mServico* e também o controle da última data de backup que ficará armazenada no documento *backup.ini*. Defina a propriedade *Interval* do *TimerBackup* para 3600000 (três milhões e seiscentos mil) representando um intervalo de verificação de *1h30min*.

Na codificação do evento *OnTimer* do componente criaremos as regras da verificação, onde iremos ler o arquivo das configurações, realizar a comparação de datas e se for necessário, realizar o backup da lista. Acessando a página de códigos do serviço, adicione primeiramente as *Units IniFiles* e *SWSystem*.

A *Unit IniFiles* contém as funções e classes necessárias para se trabalhar com arquivos do tipo *INI*. Em seguida declare uma *procedure* chamada *Backup* recebendo como parâmetro os valores *Compactar, Tipo, CaminhoArquivo* e *Destino*, todas do tipo *String*, assim como podemos ver a seguir:

```
procedure Backup(Compactar, Tipo,
CaminhoArquivo, Destino: string)
```

Essa *procedure* será chamada no momento da cópia e compactação dos arquivos. Utilize as teclas *Shift + Ctrl + C* para criar o cabeçalho do procedimento e adicione a ela o código da **Listagem 6**, onde temos um método que receberá alguns parâmetros de entrada utilizados nas verificações e desenvoltura do processo de backup. O código é muito semelhante ao visto no desenvolvimento da aplicação *client* onde primeiramente verificamos à necessidade de compactação. Em caso negativo, faz-se apenas a cópia dos arquivos compactando-os ou não dependendo do que foi selecionado anteriormente. Definido o método de compactação, uma linha de comando é configurada e executada logo após de uma chamada de comandos externos. Buscamos o caminho absoluto do respectivo compactador seguido por parâmetros que farão a compactação caso necessário. Simplesmente executamos o método *CopyFiles* do Delphi.

Finalizada a codificação do procedimento, faremos agora a implementação do evento que irá disparar o procedimento passando-lhe todos os parâmetros necessários para execução da tarefa. No evento responsável pela execução e configuração dos parâmetros, adicionaremos ao evento *OnTimer* do *TimerBackup* a leitura dos arquivos de configuração (arquivos **.ini*) e também do arquivo que contém a lista de arquivos a serem copiados no backup, para que os mesmos sejam carregados e organizados.

Adicionando ao evento *OnTimer* o código da **Listagem 7,** observe que após a declaração das variáveis estamos criando um objeto do tipo *TIniFile* passando como parâmetro o endereço

Listagem 6. Procedure que executará o backup

```
procedure TmServico.Backup(Compactar, Tipo, CaminhoArquivo,
  Destino: string);
var
  NomeArquivo: string;
  Temp: string;
  LinhadeComando: string;
begin
  if Compactar = 'S' then
  begin
    if Tipo = 'Z' then
    begin
      Temp := ChangeFileExt(CaminhoArquivo, '.rar');
      NomeArquivo := Destino + ExtractFileName(temp);
      LinhadeComando :=
        'C:\Program Files (x86)\WinRAR\WINRAR.EXE a "' +
        NomeArquivo + '" ' + CaminhoArquivo ;
    end
    else
    begin
      Temp := ChangeFileExt(CaminhoArquivo, '.zip');
      NomeArquivo := Destino + ExtractFileName(temp);
      LinhadeComando := 'C:\Program Files (x86)\WinZip\WINZIP32.EXE -A "'
        + NomeArquivo + '" ' + CaminhoArquivo;
    end;
    try
      WinExec(Pchar(LinhadeComando),1);
    except
    end;
  end
  else
  begin
    Destino := Destino + ExtractFileName(CaminhoArquivo);
    if not CopyFile(PChar(CaminhoArquivo), PChar(Destino), true) then
      ShowMessage('Erro ao copiar ' + CaminhoArquivo + ' para ' + Destino);
  end;
end;
```

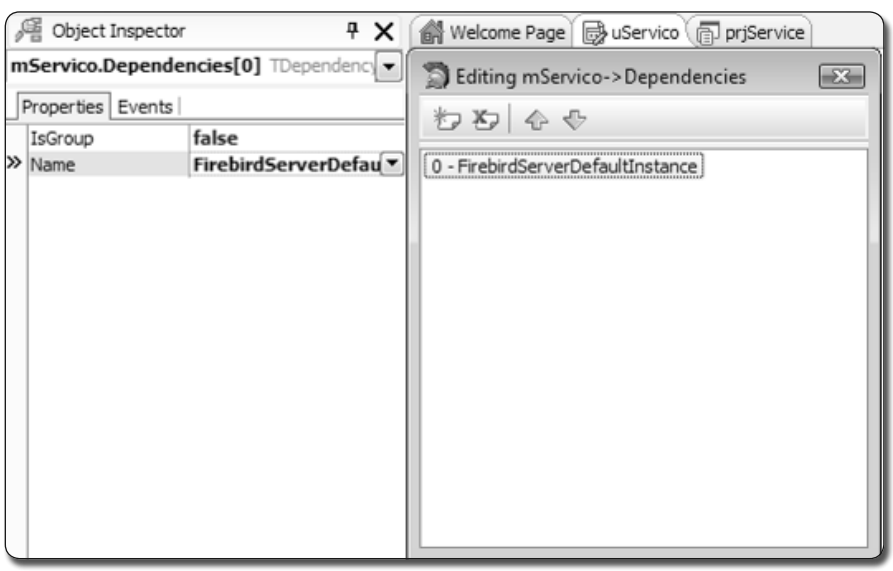


Figura 3. Definindo serviço do Firebird como dependência

Listagem 7. Código para start do backup

```
procedure TmServico.TimerBackupTimer(Sender: TObject);
var
  conf: TIniFile;
  pData: TDate;
  Compactar: string;
  TipoComp: string;
  Destino: string;
  Lista: TStringList;
  I: Integer;
  CaminhoArquivo: string;
begin
  conf := nil;
  Compactar := 'N';
  try
    try
      conf := TIniFile.Create(gsAppPath+'backup.ini');
      pData := StrToDate(conf.ReadString('BACKUP','ULTIMO',''));
      if pData < Date then
      begin
        Compactar := Trim(conf.ReadString('BACKUP','COMPACTAR',''));
        if Compactar = 'S' then
          TipoComp := Trim(conf.ReadString('BACKUP','TIPO',''));
        Destino := conf.ReadString('BACKUP','DESTINO','');
        Lista := TStringList.Create;
        Lista.LoadFromFile(gsAppPath+'lista.ini');
        if Compactar = 'S' then
        begin
          for I := 0 to Lista.Count - 1 do
            CaminhoArquivo := '"' + Lista.Strings[I] +
              '" ' + CaminhoArquivo;
          Backup(Compactar,TipoComp,CaminhoArquivo,Destino);
        end
        else
        begin
          for I := 0 to Lista.Count - 1 do
          begin
            CaminhoArquivo := Lista.Strings[I];
            Backup(Compactar,TipoComp,CaminhoArquivo,Destino);
          end;
        end;
        conf.WriteString('BACKUP','ULTIMO',
          FormatDateTime('dd/MM/yyyy',Date));
      end;
    except
    end;
  finally
    conf.Free;
    Lista.Free;
  end;
end;
```

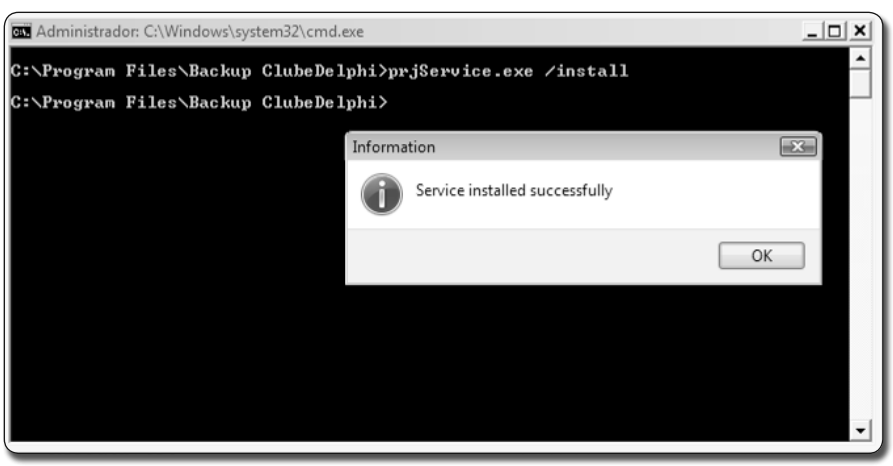


Figura 4. Instalação de serviço do Windows

do arquivo *backup.ini*. Declare também as *Unit IniFiles* e *SWSystem* que serão necessárias para o reconhecimento de ambos os objetos.

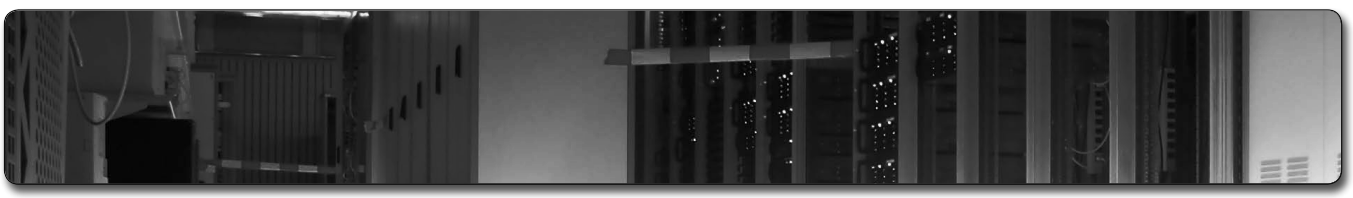
Para a variável *pData*, veja que realizamos a leitura do valor armazenado no arquivo de configurações e o comparamos a data atual do sistema. No caso da data atual ser superior a armazenada, a chamada ao backup será executada. O código inicia-se pelo carregamento da lista de arquivos para a cópia e posterior montagem da linha de comando. Para os casos onde a compactação não será necessária, uma contagem da quantidade de arquivos é realizada e por meio do uso de um comando *for* a cópia é realizada. Finalizado o processo de backup, reescrevemos a data do último serviço nas configurações, fazendo com que para este mesmo dia, nenhuma cópia a mais seja executada.

Instalando o serviço

Ao contrário das instalações de sistemas normais como você esta acostumando, com telas dinâmicas, botões de *Next* ou *Avançar*, a instalação de serviços é realizada através do *Prompt de Comandos* do *Windows*, por meio de linha de comando. Antes de iniciarmos a instalação de fato, vá até o diretório *C:\Arquivos de Programas* e crie uma pasta com o nome *Backup ClubeDelphi*. Copie para esta os executáveis da aplicação cliente (*prjBackup.exe*), o executável da *Service Application* (*prjService.exe*), além, é claro, dos dois arquivos de texto contendo as configurações que ambos os executáveis (*backup.ini* e *lista.ini*).

Abra o *Prompt de Comando* do *Windows* digitando *cmd* no *Iniciar>Executar*. Entre no diretório que acabamos de criar no *C:\Arquivos de Programas*. Em seguida digite:

```
C:\>prjService.exe /install
```



Conforme exibido na **Figura 4** obtemos a mensagem de o serviço foi instalado com sucesso. A instalação ainda não significa que o serviço esteja rodando, e para tanto precisaremos acessar suas configurações e dar o *Start* inicial ao processo.

> **Nota**: Para desinstalar o serviço, basta primeiramente pará-lo no ícone *Serviços* que encontra-se dentro de *Ferramentas Administrativas* no *Painel de Controle*. Em seguida, abra novamente o *prompt* e entre na pasta onde o serviço está instalado. Basta então digitar o comando */uninstall* seguido do nome do projeto.

Configurando e iniciando o serviço

Para dar start na aplicação e com isso fazê-la funcionar, basta entrar no *Painel de Controle>Ferramentas Administrativas>Serviços* e localizar o nosso serviço que deverá aparecer com o nome que configuramos no projeto, *Serviço de Backup ClubeDelphi*. Ao encontrar a entrada, utilize o botão direito sobre o mesmo e acesse suas propriedades, observando que várias opções de configuração ainda estão disponíveis. Na guia *Logon* temos a opção de *Permitir que o serviço interaja com a área de trabalho*, que seria útil no caso de exibirmos alguma mensagem ou mesmo se tivéssemos a construção de algum formulário em tempo de execução. Não estando habilitada, esta opção rejeita qualquer solicitação do serviço para exibir informações na tela do usuário, permanecendo apenas como processo oculto em execução.

Retornando a guia *Geral*, mantenha o tipo de inicialização como *Automática*, fazendo com que o serviço seja iniciado automaticamente a cada inicialização do Sistema Operacional e em seguida, inicie o processo com um clique sobre o botão *Iniciar*. Pronto, seu serviço de backup já estará em execução no seu computador. Utilize agora o sistema cliente para as configurações necessárias e lista de arquivos do backup e aguarde o período definido no *Timer* para que o primeiro backup seja realizado.

Por *default*, os backups serão realizados em modo *background*, ou seja, não visíveis ao usuário.

Conclusão

O desenvolvimento de aplicações *Service Application* é ideal para que possamos automatizar tarefas. O exemplo aqui realizado é só um dos muitos aplicativos que podemos desenvolver usando esse recurso. Uma boa idéia é criar serviços para os mais variados tipos de tarefas, como fazer a atualização automática do software via internet, como fazem a maiores dos anti-vírus, por exemplo.

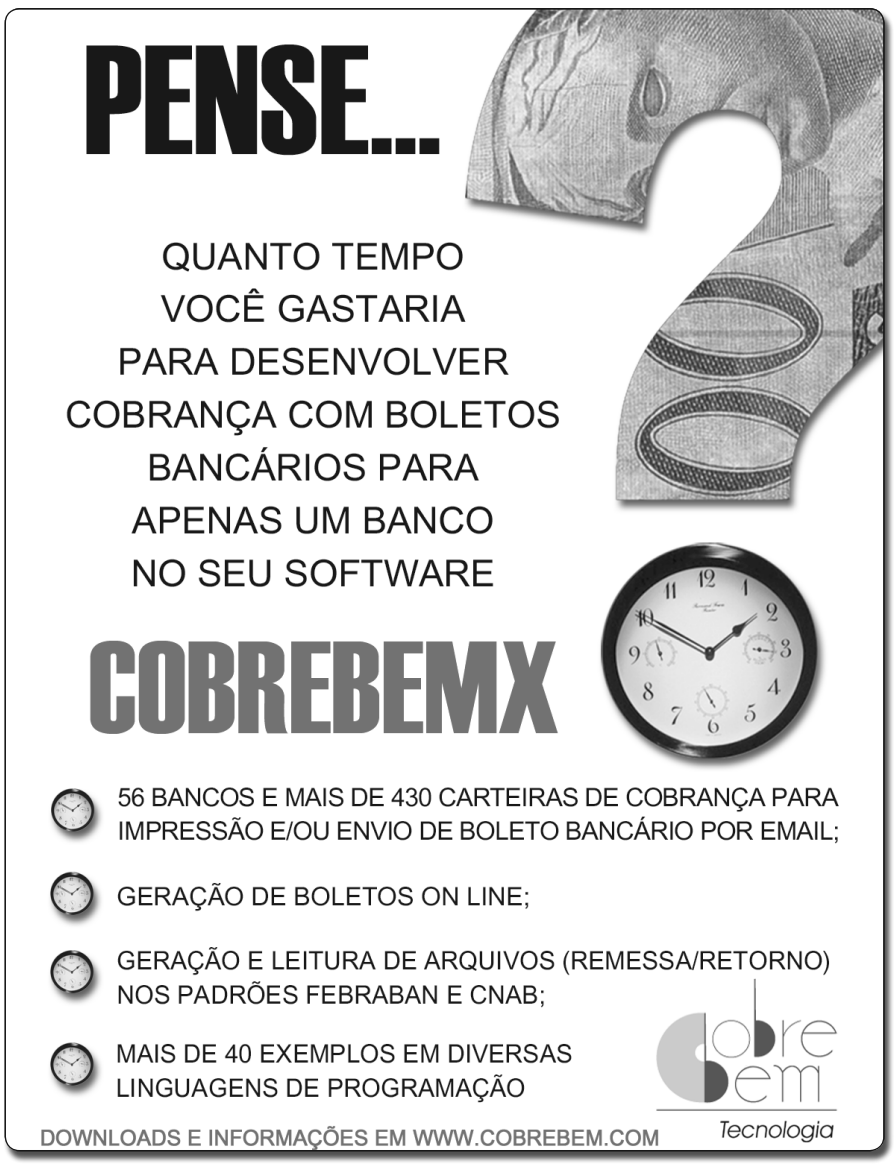
Enfim, cabe ao leitor aprimorar seus conhecimentos montando novas aplicações, criando idéias e estudando mais detalhes sobre a criação de serviços. Abraços e até a próxima. ●

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

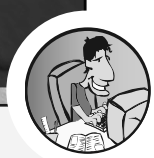
www.devmedia.com.br/clubedelphi/feedback



Nesta seção você encontra artigos para iniciantes na linguagem Delphi

Escrevendo e trabalhando com Dlls

Saiba como desenvolver e usar dlls no Delphi



Resumo DevMan

Nem sempre um sistema trabalha sozinho, aliás, diria que nunca trabalha sozinho. Quando geramos um produto em Delphi, estamos criando um aplicativo que depende, ligeiramente, de outros programas para que funcione corretamente no sistema operacional. Algumas dessas dependências são as DLL's. O Windows está repleto delas por todo o sistema.

O interessante disso tudo, é que tanto o sistema operacional quanto nosso próprio sistema podem fazer uso de DLL para encapsular determinadas funcionalidades do software. Veremos nesse artigo como criar uma DLL e usá-la na prática no dia-a-dia.

Nesse artigo veremos

- Desenvolvimento de DLL;
- Carregamento implícito;
- Carregamento explícito;

Qual a finalidade

- Demonstrar como fazer o desenvolvimento de DLL'S, como utilizá-las e onde aplicá-las;

Quais situações utilizam esses recursos?

- Há situações em que é necessário compartilhar determinadas funcionalidades entre mais de um produto. Com o desenvolvimento de dlls, isso é perfeitamente possível;

Algumas das minhas grandes dúvidas no início de minha carreira como desenvolvedor Delphi, certamente foram: O que são *dll's*, como criá-las, como funcionam e onde aplicá-las? Realmente o tema sempre foi pouco discutido entre os desenvolvedores e até hoje conheço veteranos que tem certa dificuldade em trabalhar com esse formato de arquivo. Mas na verdade desenvolver e usar *dll* no dia-a-dia é mais fácil do que se pensa.

Para os iniciantes, *DLL* significa *Dynamic Link Library* ou *Biblioteca de Linkedição Dinâmica*. Mas o que vem a ser isso? Bem, precisamos entender como funciona exatamente um programa, um software para entendermos as DLL's. Dessa forma conseguiremos entender bem o papel delas no dia-a-dia.

O computador, em se tratando de *hardware* e como é conhecido de todos, possui uma coisa chamada memória *RAM* ou *Random Access Memory – Memória de*

Adriano Santos

(falecom@adrianosantos.pro.br)

é desenvolvedor Delphi desde 1998, professor e programador PHP, bacharel em Comunicação Social pela Universidade Cruzeiro do Sul, SP, é editor geral das revistas ClubeDelphi e WebMobile. Mantém o blog Delphi to Delphi (www.delphitodelphi.blogspot.com) com dicas, informações e tudo sobre desenvolvimento Delphi e é membro fundador do DUG-SP Delphi Users Group São Paulo (www.dug-sp.com)

Acesso Aleatório - também chamada de *Memória Volátil*. É na memória *RAM* que os programas armazenam suas informações e posteriormente (dependendo da informação) enviam para o disco rígido. Portanto, quando abrimos um software qualquer como o *Word* ou *Excel*, partes do aplicativo são gravadas nessa memória. Acontece que em determinados casos, diversos programas podem compartilhar das mesmas funcionalidades, principalmente quando fazem parte do mesmo grupo (empresa) que os desenvolveu. Exemplo: *Microsoft Word, Excel, PowerPoint, Windows* e demais produtos. Isso significa que não é necessário termos várias instâncias do mesmo objeto em memória para executar a mesma funcionalidade. Por isso, muitas vezes se faz necessário a criação de *dll's* para que possamos disponibilizar uma ou mais funções que podem ser usadas por mais de um sistema. Esse é o principal propósito desses arquivos.

Em uma das edições passadas da *ClubeDelphi*, falei sobre *API's* (*Application Programming Interface*) e esse é um exemplo claro de como e onde usar *dll's*. Nesse artigo veremos como desenvolver uma *DLL*, as diferentes formas de carregamento e como utilizar as funções que nós mesmos desenvolveremos. Veremos o quanto é fácil integrar nosso sistema a esse tipo de arquivo e como eles podem nos ajudar no dia-a-dia. Mão na massa!

Entendendo melhor as DLL's

Acredito que uma das vantagens mais importantes do uso de *DLL* chama-se: *encapsulamento*. Vimos em outros artigos que *encapsular* significa isolar determinadas partes do software em *um componente*, *uma classe* ou mesmo em uma *DLL*. Isso porque não precisamos (do ponto de vista do programador/codificador) saber exatamente como que funciona tal mecanismo. Precisamos apenas saber como chamar e quais parâmetros, quando necessário, passar para a função ou método desejado. Em outras palavras, basta que saibamos da existência do método e de seus requisitos para execução. Exemplificando melhor: quando precisamos executar um programa externo via codificação *Delphi*, podemos simplesmente utilizar a função *WinExec* presente na *Unit Windows* que por sua vez faz referência a *DLL* do *Kernel32.dll* do sistema operacional. Para que *WinExec* execute o programa que desejamos, passamos para ela dois parâmetros, que são:

- *lbCmdLine:* caminho completo do programa;
- *uCmdShow:* como o programa será aberto.

A linha de comando utilizada para executar a calculadora do *Windows*, por exemplo, é:

```
WinExec('C:\Windows\System32\Calc.exe',
  SW_SHOWNORMAL);
```

Muito bom, mas você sabe me dizer exatamente o que o método *WinExec* faz para executar um programa? Não é possível saber, isso porque a função *WinExec* está *encapsulada* na *DLL Kernel32.dll* como mencionei anteriormente. Mesmo porque, qual seria nossa necessidade em saber isso? Praticamente nenhuma, o mais importante é entendermos e sabermos como usar essa funcionalidade da *DLL*.

Em nosso exemplo, faremos o desenvolvimento de uma DLL para uso em nosso sistema. Colocaremos algumas funções e as usaremos na prática.

Nota do DevMan

Application Programming Interfaces são arquivos DLL que possuem métodos e funções em comum que podem ser utilizadas por qualquer linguagem de programação. Isso significa que quando pedimos ao Windows para exibir as propriedades de um determinado arquivo, uma função está sendo chamada e essa mesma função pode se invocada usando qualquer linguagem de programação.

Listagem 1. Criando uma DLL

```
library Project1;
{ Important note about DLL memory management: ShareMem must be the
  first unit in your library's USES clause AND your project's (select
  Project-View Source) USES clause if your DLL exports any procedures or
  functions that pass strings as parameters or function results. This
  applies to all strings passed to and from your DLL--even those that
  are nested in records and classes. ShareMem is the interface unit to
  the BORLNDMM.DLL shared memory manager, which must be deployed along
  with your DLL. To avoid using BORLNDMM.DLL, pass string information
  using PChar or ShortString parameters. }
uses
  SysUtils,Classes;
{$R *.res}
begin
end.
```

Construindo a DLL

A primeira coisa que devemos saber é como funciona a estrutura de desenvolvimento de uma DLL. Vejamos na prática como isso funciona. Crie uma nova aplicação utilizando o *Template* para criação de DLL's. Você encontrará isso no menu *File>New>Other>New>DLL Wizard* no *Delphi 7* ou *File>New>Other>Delphi Projects>DLL Wizard* em versões superiores, como o *RAD Studio 2007*, por exemplo. O projeto então será criado e um código semelhante a **Listagem 1** será automaticamente gerado pelo *Delphi*.

Perceba que há uma diferença bastante grande se compararmos a um projeto de software utilizando a opção *File>New>Application*, usado para criar aplicações *Win32*. Aqui vemos que não temos um *form* e ao invés de encontrarmos *program* na primeira linha do programa, o que vem escrito é *library*, indicando que o projeto é uma *DLL*. Além disso, é claro, há outras diferenças como a ausência das sessões *Interface* e *Implementation*.

Muito bem, vamos construir uma *DLL* que contenha alguns métodos para que possamos entender como funciona o seu uso. Vamos criar métodos para converter textos em maiúsculas ou minúsculas conforme a necessidade e mais uma função para converter números inteiros em texto. É claro que esses métodos já existem no próprio Delphi, mas vamos encarar como uma novidade em nosso sistema.

A primeira coisa que devemos fazer é inserir o código da **Listagem 2** logo após a diretiva de compilação *{$R *.res}*. Certifique-se de que estará acima do *begin...end* ao final da *Unit*. Se tirarmos o bloco de comentário gerado pelo Delphi, nossa *DLL* ficará como na **Listagem 3**.

Repare que temos algumas novidades no código em relação a aplicações *Win32*. Uma delas é o retorno das funções. Estamos utilizando *ShortString* ao invés de *String* como de costume. Isso é uma medida adotada para compatibilizar nossa *DLL* com outras linguagens de programação. Em C, por exemplo, não existem tipos de *string* longos como no *Delphi*. Então, se algum programador C precisar utilizar nossa *DLL* não terá problemas de incompatibilidade.

A segunda novidade é a palavra reservada *exports*. Perceba que declaramos nessa sessão as três funções que criamos. É nesse momento que informamos a *DLL* quais funções estão acessíveis ao mundo externo. Veja, que criamos também uma *procedure* chamada *MensagemConfirmacao* que é chamada sempre que um método é executado. Essa *procedure* não possui a palavra reservada *stdcall* e também não foi declarada em *exports*, ou seja, não será enxergada por processos externos.

Crie um diretório em seu micro e salve o projeto com o nome *MinhaDll.dpr*. Para facilitar, vamos criar um grupo de projetos e adicionar a ele os nossos exemplos. Clique em *View>Project Manager* e você verá uma tela semelhante a **Figura 1**.

Clique com o direito em *ProjectGroup1* e *Save Project Group As...* e dê o nome de *ProjetoDLL* ou o nome que preferir. Agora ficará mais fácil trabalharmos com mais de um projeto ao mesmo tempo sem precisar abrir mais de um *Delphi* ou ficar abrindo e fechando nossos projetos. Em teoria, nossa DLL está pronta. Pressione Ctrl + F9 e veja se o *Delphi* compila normalmente nosso projeto.

Chamando a DLL implicitamente

Há basicamente dois modos de se utilizar uma *DLL*: implícito e explícito. O modo implícito se dá quando declaramos as funções/métodos a serem utilizados diretamente no software. Esse método é usado por "n" motivos e um deles é quando há necessidade obrigatória do uso da *DLL*, isso porque quando declaramos as funções a serem usadas diretamente no projeto o aplicativo carregará a biblioteca (*DLL*) no mesmo instante e a manterá em memória. Por isso necessitamos que ela esteja no mesmo diretório do executável final de nossa aplicação.

Em outras palavras, o método implícito carrega a *DLL* para memória e a deixa lá até que o sistema que a usa seja fechado. Se não houver a necessidade do arquivo manter-se carregado o tempo todo, esse método deve ser desprezado. Vejamos como fazer uso desse método para carregar e utilizar os métodos criados em nosso projeto.

A primeira coisa é criarmos um projeto dentro de nosso grupo. Para isso clique com o botão direto no nome de nosso grupo no *Project Manager* e selecione *Add New Project*. Na aba *New* escolha *Application*. Desenhe uma tela semelhante a **Figura 2**.

Nós temos alguns *Labels* para dar nomes as controles *Edit* e *SpinEdit* em tela. E mais três *buttons*. Se preferir troque os nomes dos componentes, mas nesse exemplo mantive todos com os nomes *default*. Os

Listagem 2. Código das funções da DLL

```
procedure MensagemConfirmacao(AMensagem: WideString);stdcall;
begin
  MessageDlg(AMensagem, mtInformation, [mbOk], 0);
end;
function MeuIntToStr(Numero: Integer): ShortString;stdcall;
begin
  Result := IntToStr(Numero);
  MensagemConfirmacao('Convertido com sucesso!');
end;
function MeuUpperCase(s: ShortString): ShortString; stdcall;
begin
  Result := UpperCase(s);
  MensagemConfirmacao('Texto convertido em Maiúsculas!');
end;
function MeuLowerCase(s: WideString): ShortString; stdcall;
begin
  Result := LowerCase(s);
  MensagemConfirmacao('Texto convertido em Minúsculas!');
end;
exports
  MeuUpperCase, MeuLowerCase, MeuIntToStr;
```

Listagem 3. Código completo da DLL

```
library MinhaDLL;
uses
  SysUtils, Classes;

{$R *.res}
procedure MensagemConfirmacao(AMensagem: WideString);stdcall;
begin
  MessageDlg(AMensagem, mtInformation, [mbOk], 0);
end;
function MeuIntToStr(Numero: Integer): ShortString;stdcall;
begin
  Result := IntToStr(Numero);
  MensagemConfirmacao('Convertido com sucesso!');
end;
function MeuUpperCase(s: ShortString): ShortString; stdcall;
begin
  Result := UpperCase(s);
  MensagemConfirmacao('Texto convertido em Maiúsculas!');
end;
function MeuLowerCase(s: WideString): ShortString; stdcall;
begin
  Result := LowerCase(s);
  MensagemConfirmacao('Texto convertido em Minúsculas!');
end;
exports
  MeuUpperCase, MeuLowerCase, MeuIntToStr;
end.
```

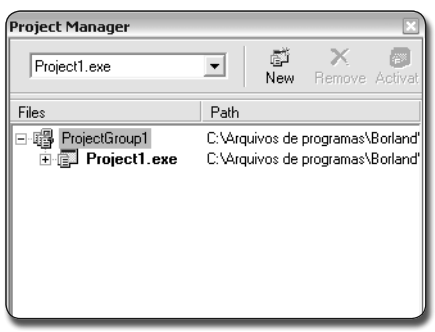


Figura 1. Project Manager

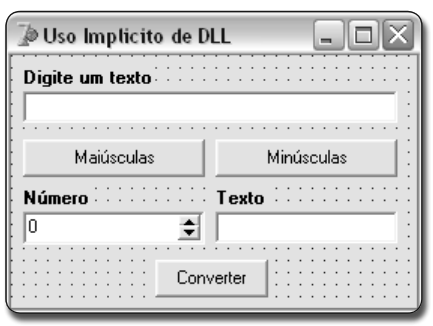


Figura 2. Exemplo de uso da DLL implícitamente

dois primeiros botões (*Maiúsculas* e *Minúsculas*) farão uma chamada as métodos MeuUpperCase e MeuLowerCase, para transformar o texto do primeiro *Edit* em maiúsculas ou minúsculas. Já o terceiro botão fará uma chamada ao método *MeuIntToStr* e enviará o número convertido para o segundo *Edit*.

Para que possamos utilizar as funções mencionadas, teremos que declará-las no corpo da *Unit* de nosso *form*. Localize então a palavra reservada *Implementation* do formulário e logo acima da declaração do *form* digite o código da **Listagem 4**.

Aqui estamos informando ao executável, quem é a *DLL* que possui tal método e qual o nome dessa função. A digitação errada do *name* do método pode causar um erro de *Access Violation*, já que o executável não consegue encontrar a função declarada. Após isso, precisamos apenas fazer a chamada normalmente no evento *OnClick* de cada botão. Veja o código da **Listagem 5** atentamente, e implemente a chamada para cada *Button* em tela. Em seguida, compile o projeto e execute-o. Procure testar cada funcionalidade e verificar se está tudo de acordo.

Para provar que a *DLL* foi carregada para a memória, faça um teste. Abra o sistema e em seguida abra o *Windows Explorer* ou *Meu Computador* e navegue até a pasta em que salvou o projeto e a *DLL*. Tente apagar a *DLL* com o programa aberto. Verá que o *Windows* exibirá uma mensagem informando que não é possível excluir o arquivo, pois possivelmente está em uso.

Outro teste interessante de se fazer é apagar a *DLL* e então tentar abrir o sistema. Um erro como na **Figura 4** será exibido. Isso acontece devido ao que informei antes. Como estamos usando-a implicitamente em nosso projeto, o sistema a carrega logo que entra no ar.

Listagem 4. Declaração das funções da DLL no projeto

```
[...]
function MeuUpperCase(s: ShortString): ShortString; stdcall;
  external 'MinhaDll.dll' name 'MeuUpperCase';
function MeuLowerCase(s: WideString): ShortString; stdcall;
  external 'MinhaDll.dll' name 'MeuLowerCase';
function MeuIntToStr(Numero: Integer): ShortString;stdcall;
  external 'MinhaDll.dll' name 'MeuIntToStr';
var
  Form1: TForm1;
[...]
A novidade aqui é que adicionamos o trecho a seguir depois da palavra reservada stdcall
de cada função.
external 'MinhaDll.dll' name 'MeuUpperCase';
```

Listagem 5. Código dos botões de tela

```
procedure TForm1.Button4Click(Sender: TObject);
begin
  Edit1.Text := MeuUpperCase(Edit1.Text);
end;
procedure TForm1.Button5Click(Sender: TObject);
begin
  Edit1.Text := MeuLowerCase(Edit1.Text);
end;
procedure TForm1.Button3Click(Sender: TObject);
begin
  Edit2.Text := MeuIntToStr(SpinEdit1.Value);
end;
```

Chamando a DLL explicitamente

O modo explícito de se fazer o carregamento da *DLL* exige um pouco mais de explicações, pois faremos todo o processo dinamicamente. Podemos dizer que em determinadas situações é mais vantajoso usar esse método, pois a *DLL* não aloca espaço em memória desnecessário. Outra grande vantagem diz respeito a distribuição. É perfeitamente possível fazer a exclusão da biblioteca com o programa em execução, ou seja, caso você possua um programa que faz a atualização automática de versão de sua *DLL*, o programa que utiliza tal arquivo não precisa necessariamente ser fechado, a menos que esteja usando alguma funcionalidade da *DLL*.

O carregamento dinâmico é feito utilizando três *API's* do *Windows* presentes na *DLL Kernel32.dll*, são elas:

- *LoadLibrary:* responsável por carregar a *DLL* para a memória;
- *GetProcAddress:* recupera o endereço de memória da função que precisamos usar;
- *FreeLibrary:* elimina da memória a *DLL* carregada.

A título de curiosidade, a declaração de cada uma pode ser vista a seguir e estão declaradas na *Unit Windows* do *Delphi*.

```
function LoadLibrary;
  external kernel32 name 'LoadLibraryA';
function GetProcAddress;
  external kernel32 name 'GetProcAddress';
function FreeLibrary;
  external kernel32 name 'FreeLibrary';
```

Para que tenhamos um bom parâmetro de comparação entre os dois métodos, faça um clone da aplicação que criamos anteriormente. Clique novamente com o botão direito no grupo de projetos no *Project Manager* e escolha *Add New Project*. Salve o novo projeto como *prjExplicito.dpr* ou o nome que preferir. Abra o formulário do primeiro projeto que desenvolvemos, copie todos os objetos e cole-os no formulário desse novo projeto.

A primeira medida que devemos tomar é criar três classes que farão referência as funções em nossa DLL. Faça isso digitando o código a seguir antes da sessão *type* do formulário. Veja:

Figura 3. Teste no sistema carregando a dll implicitamente

Figura 4. Erro ao tentar executar o sistema sem a presença da MinhaDll.dll no diretório

Listagem 6. Código para uso da função Maiúsculas (dinamicamente)

```
procedure TForm2.Button4Click(Sender: TObject);
var
  Handle: THandle;
  mUpperCase: TMeuUpperCase;
begin
  Handle := LoadLibrary('MinhaDLL.dll');
  if Handle <> 0 then
  begin
    mUpperCase := GetProcAddress(Handle, 'MeuIntToStr');
    Edit2.Text := mUpperCase(Edit2.Text);
    FreeLibrary(Handle);
  end;
end;
```

Listagem 7. Código do botão Minúsculas

```
procedure TForm2.Button5Click(Sender: TObject);
var
  Handle: THandle;
  mLowerCase: TMeuLowerCase;
begin
  Handle := LoadLibrary('MinhaDLL.dll');
  if Handle <> 0 then
  begin
    mLowerCase := GetProcAddress(Handle, 'MeuLowerCase');
    Edit2.Text := mLowerCase(Edit2.Text);
    FreeLibrary(Handle);
  end;
end;
```

Listagem 8. Código do botão Converter

```
procedure TForm2.Button3Click(Sender: TObject);
var
  Handle: THandle;
  mIntToStr: TMeuIntToStr;
begin
  Handle := LoadLibrary('MinhaDLL.dll');
  if Handle <> 0 then
  begin
    mIntToStr := GetProcAddress(Handle, 'MeuIntToStr');
    Edit3.Text := mIntToStr(SpinEdit1.Value);
    FreeLibrary(Handle);
  end;
end;
```

```
type
  TMeuUpperCase =
    function(s: ShortString): ShortString;
  TMeuLowerCase =
    function(s: ShortString): ShortString;
  TMeuIntToStr =
    function (Numero: Integer): ShortString;
```

Agora faremos a codificação do botão *Maiúsculas*. Clique duas vezes nele para acessarmos seu evento *OnClick* e então digite o código da **Listagem 6.** Vejamos o que temos aqui. Declaramos duas variáveis: *Handle* e *mUpperCase*. *Handle* receberá a referência da *DLL* em memória, ou seja, atribuímos a ela o resultado do método *LoadLibrary*, que é justamente o endereço de memória. Todo programa, quando iniciado(instanciado), recebe um endereço de memória que é a forma como o sistema operacional o mapeia.

A variável *mUpperCase* por sua vez, receberá o endereço de memória da função *MeuUpperCase* que está declarada em nossa variável. Isso significa, que poderemos acessar o resultado da função atribuindo-a a um objeto em tela, nesse caso a propriedade *Text* do *Edit2*. Por fim chamamos o método *FreeLibrary* para liberar de memória nossa *DLL*. O interessante disso tudo é que nossa biblioteca não permanecerá em memória depois de sua utilização. Experimente agora codificar os demais botões conforme as **Listagens 7** e **8**, sendo a *7* é do botão *Minúsculas* e *8* do *Converter*. A explicação é exatamente a mesma.

Rode a aplicação e faça o mesmo teste que foi feito com nosso projeto anterior. Tente apagar a *DLL* do diretório com o software aberto ou iniciar ele sem a presença dela. O sistema somente acusará a falta da mesma ao clicarmos em um dos botões que fazem sua chamada. Na verdade, não será exibido nenhum erro. Os botões apenas não funcionarão como se não tivéssemos programado nenhuma funcionalidade para eles.

Há algumas formas de contornar esse pequeno problema. No evento *OnClick* dos botões podemos fazer uma checagem de rotina usando o método *FileExists*, que verifica se determinado arquivo existe. Veja:

```
if not FileExists(
  ExtractFilePath(Application.ExeName)+
  'MinhaDLL.dll') then
begin
  ShowMessage('DLL não encontrada.');
  Exit;
end;
```

Podemos incluir esse código antes da chamada ao método *LoadLibrary*. Assim, caso não exista a *DLL* no diretório da aplicação o programa avisará e o fluxo do evento será parado (*Exit*).

Outra forma é incluir um *else* no *IF* no *Handle*. Se a variável *Handle* retornar 0 (zero) significa que não conseguiu carregar a *DLL* por algum motivo, então exibimos uma mensagem.

```
if Handle <> 0 then
begin
end
else
  ShowMessage(
    'Não foi possível carregar a DLL.');
```

Considerações finais

O uso de *DLL's*, como pudemos ver, não é nenhum bicho de sete cabeças. Com esse estudo abrimos brecha para o desenvolvimento de diversas soluções para os mais variados problemas do dia-a-dia e que podem ser solucionados desenvolvendo uma simples *DLL*. Um bom exemplo disso seria criar um sistema de licenciamento do software através de *DLL*. É claro, que o uso desse recurso deve ser bastante pensado para evitar maiores problemas, principalmente quando se guarda dados confidenciais dentro desse tipo de arquivo.

Conclusão

Como pudemos perceber, o uso de DLL é bastante simples e não requer nenhum conhecimento específico. O importante é entender o conceito de uso desses arquivos aplicá-lo corretamente.

Procure estudar mais o processo de desenvolvimento de DLL e quais tipos de dados são limitados no desenvolvimento. Um forte abraço e até a próxima. ●

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

10 Motivos para hospedar seu site na RedeHost!

30 dias GRÁTIS para testar.

Mais de 15 mil clientes e 6 anos no mercado.

Até 50Gb de espaço para e-mails ou site.

Planos a partir de R$ 13,90.

Webmail em Ajax com calendário, tarefas, anotações e RSS.

ASP, PHP, .net e MySQL no mesmo plano.

Domínios adicionais ilimitados (apontam para o site principal).

Data center no Brasil com Links Redundantes.

Infra-estrutura com servidores Dell e redes Cisco.

Estatísticas de acesso com 75 relatórios.

RedeHost

www.redehost.com.br

A INVASÃO DE PROPRIEDADE NÃO SE LIMITA A RESIDÊNCIAS.

Assegure-se que seu software está protegido pelo líder em segurança da informação, antes que seja tarde demais.

SafeNet tem à sua disposição a proteção de software mais avançada e segura. As Chaves de Hardware Sentinel oferecem tecnologia que incorpora criptografia de chave pública, criptografia AES e autenticação interna para garantir o mais alto nível de proteção anti-pirataria.

Peça seu Kit de Desenvolvimento hoje mesmo.

Entre em contato em +55 11 4208-7700 em SP-Brasil
ou por e-mail infopirataria@safenet-inc.com

25 YEARS
1983–2008

SafeNet